FORM PTO-1390 (Modified)   U.S. DEPARTMENT OF COMMERCE PATENT AND TRADEMARK OFFICE

TRANSMITTAL LETTER TO THE UNITED STATES DESIGNATED/ELECTED OFFICE (DO/EO/US) CONCERNING A FILING UNDER 35 U.S.C. 371

FEB 2 2 2002

ATTORNEY'S DOCKET NUMBER
P-6451
U.S. APPLICATION NO. (IF KNOWN, SEE 37 CFR
10/069670

INTERNATIONAL APPLICATION NO.	INTERNATIONAL FILING DATE	PRIORITY DATE CLAIMED
PCT/FR00/02349	21 August 2000	23 August 1999

TITLE OF INVENTION
PROTOCOL FOR MANAGING AND METHOD OF VERIFYING AND OF CONVERTING A DOWNLOADED PROGRAM FRAGMENT, AND CORRESPONDING SYSTEMS

APPLICANT(S) FOR DO/EO/US
Xavier LEROY

Applicant herewith submits to the United States Designated/Elected Office (DO/EO/US) the following items and other information:

1. ☒ This is a **FIRST** submission of items concerning a filing under 35 U.S.C. 371.
2. ☐ This is a **SECOND** or **SUBSEQUENT** submission of items concerning a filing under 35 U.S.C. 371.
3. ☐ This is an express request to begin national examination procedures (35 U.S.C. 371(f)) at any time rather than delay examination until the expiration of the applicable time limit set in 35 U.S.C. 371(b) and PCT Articles 22 and 39(1).
4. ☒ A proper Demand for International Preliminary Examination was made by the 19th month from the earliest claimed priority date.
5. ☒ A copy of the International Application as filed (35 U.S.C. 371 (c) (2))
    a. ☒ is transmitted herewith (required only if not transmitted by the International Bureau).
    b. ☐ has been transmitted by the International Bureau.
    c. ☐ is not required, as the application was filed in the United States Receiving Office (RO/US).
6. ☒ A translation of the International Application into English (35 U.S.C. 371(c)(2)).
7. ☒ A copy of the International Search Report (PCT/ISA/210).
8. ☐ Amendments to the claims of the International Application under PCT Article 19 (35 U.S.C. 371 (c)(3))
    a. ☐ are transmitted herewith (required only if not transmitted by the International Bureau).
    b. ☐ have been transmitted by the International Bureau.
    c. ☐ have not been made; however, the time limit for making such amendments has NOT expired.
    d. ☐ have not been made and will not be made.
9. ☐ A translation of the amendments to the claims under PCT Article 19 (35 U.S.C. 371(c)(3)).
10. ☒ An oath or declaration of the inventor(s) (35 U.S.C. 371 (c)(4)).
11. ☒ A copy of the International Preliminary Examination Report (PCT/IPEA/409).
12. ☐ A translation of the annexes to the International Preliminary Examination Report under PCT Article 36 (35 U.S.C. 371 (c)(5)).

Items 13 to 18 below concern document(s) or information included:

13. ☐ An Information Disclosure Statement under 37 CFR 1.97 and 1.98.
14. ☒ An assignment document for recording. A separate cover sheet in compliance with 37 CFR 3.28 and 3.31 is included.
15. ☐ A **FIRST** preliminary amendment.
    A **SECOND** or **SUBSEQUENT** preliminary amendment.
16. ☐ A substitute specification.
17. ☐ A change of power of attorney and/or address letter.
18. ☒ Certificate of Mailing by Express Mail
19. ☒ Other items or information:

    PCT Request Form PCT/RO/101 (4 pgs.)
    14 Sheets of Formal Drawings
    Change of Applicant's Address (PCT/IB/306)
    Return Receipt Postcard

PCTUS1/REV03

U.S. APPLICATION NO. (IF KNOWN, SEE 37 CFR 10/069670	INTERNATIONAL APPLICATION NO. PCT/FR00/02349	ATTORNEY'S DOCKET NUMBER P-6451

20. The following fees are submitted:.		CALCULATIONS PTO USE ONLY
BASIC NATIONAL FEE (37 CFR 1.492 (a) (1) - (5)) :		
☒ Search Report has been prepared by the EPO or JPO	$890.00	
☐ International preliminary examination fee paid to USPTO (37 CFR 1.482)	$720.00	
☐ No international preliminary examination fee paid to USPTO (37 CFR 1.482) but international search fee paid to USPTO (37 CFR 1.445(a)(2))	$790.00	
☐ Neither international preliminary examination fee (37 CFR 1.482) nor international search fee (37 CFR 1.445(a)(2) paid to USPTO	$1,070.00	
☐ International preliminary examination fee paid to USPTO (37 CFR 1.482) and all claims satisfied provisions of PCT Article 33(2)-(4)	$98.00	

	ENTER APPROPRIATE BASIC FEE AMOUNT =	$890.00	

Surcharge of **$130.00** for furnishing the oath or declaration later than ☐ 20 ☐ 30 months from the earliest claimed priority date (37 CFR 1.492 (e)).			$0.00	

CLAIMS	NUMBER FILED	NUMBER EXTRA	RATE		
Total claims	27 - 20 =	7	x $18.00	$126.00	
Independent claims	6 - 3 =	3	x $84.00	$252.00	
Multiple Dependent Claims (**check if applicable**).			☒	$280.00	

	TOTAL OF ABOVE CALCULATIONS =	$1,548.00	

Reduction of 1/2 for filing by small entity, if applicable. Verified Small Entity Statement must also be filed (Note 37 CFR 1.9, 1.27, 1.28) (**check if applicable**).	☐	$0.00	

	SUBTOTAL =	$1,548.00	

Processing fee of **$130.00** for furnishing the English translation later than ☐ 20 ☐ 30 months from the earliest claimed priority date (37 CFR 1.492 (f)).	+	$0.00	

	TOTAL NATIONAL FEE =	$1,548.00	

Fee for recording the enclosed assignment (37 CFR 1.21(h)). The assignment must be accompanied by an appropriate cover sheet (37 CFR 3.28, 3.31) (**check if applicable**).	☒	$40.00	

	TOTAL FEES ENCLOSED =	$1,588.00	

	Amount to be: refunded	$
	charged	$

☒   A check in the amount of **$1,588.00**   to cover the above fees is enclosed.

☐   Please charge my Deposit Account No.   in the amount of   to cover the above fees.
      A duplicate copy of this sheet is enclosed.

☐   The Commissioner is hereby authorized to charge any fees which may be required, or credit any overpayment
      to Deposit Account No.   A duplicate copy of this sheet is enclosed.

NOTE: **Where an appropriate time limit under 37 CFR 1.494 or 1.495 has not been met, a petition to revive (37 CFR 1.137(a) or (b)) must be filed and granted to restore the application to pending status.**

SEND ALL CORRESPONDENCE TO:

Michael L. Kenaga, Esq.
PIPER MARBURY RUDNICK & WOLFE
P.O. Box 64807
Chicago, Illinois, 60664-0807

(312) 368-4000

28465
PATENT TRADEMARK OFFICE

SIGNATURE

Michael L. Kenaga
NAME

34,639
REGISTRATION NUMBER

February 27, 2002
DATE

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Application No. :

U.S. National Serial No. :

Filed :

PCT International Application No. :   PCT/FR00/02349

VERIFICATION OF A TRANSLATION

I, Susan POTTS BA ACIS

Director to RWS Group plc, of Europa House, Marsham Way, Gerrards Cross, Buckinghamshire, England declare:

That the translator responsible for the attached translation is knowledgeable in the French language in which the below identified international application was filed, and that, to the best of RWS Group plc knowledge and belief, the English translation of the international application No. PCT/FR00/02349 is a true and complete translation of the above identified international application as filed.

I hereby declare that all the statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the patent application issued thereon.

Date: January 28, 2002

Signature of Director :

For and on behalf of RWS Group plc

Post Office Address :   Europa House, Marsham Way,

Gerrards Cross, Buckinghamshire,

England.

[continued on next page]

As printed

(54) Title: MANAGEMENT PROTOCOL, METHOD FOR VERIFYING AND TRANSFORMING A DOWNLOADED PROGRAMME FRAGMENT AND CORRESPONDING SYSTEMS

(54) Titre: PROTOCOLE DE GESTION, PROCEDE DE VERIFICATION ET DE TRANSFORMATION D'UN FRAGMENT DE PROGRAMME TELECHARGE ET SYSTEMES CORRESPONDANTS
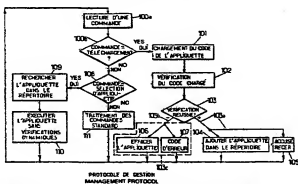
(57) Abstract: The invention relates to a management protocol and to a method for verifying a programme fragment, or applet, which has been downloaded onto a portable system. An applet downloading command (100a, 100b) is executed. Once a positive response has been received, the object code of the applet is read (101) and subjected to (102) a verification process. instruction by instruction. The verification process consists of a stage comprising the initialisation of the type stack and table of register types representing the state of the virtual machine of the portable system at the start of the execution of the applet code; and a verification, instruction by instruction, for each target current instruction, of the existence of a target branch instruction, a target exception handler call or a target sub-routine call, the effect of the instruction on the type stack and the table of register types being verified and updated. If the verification is successful (103a), the applet is registered (104) and an acknowledgement is sent (105) to the downloading drive. Otherwise, the applet is destroyed (106). The invention is suitable for use for portable systems in a Java environment.

[continued on next page]

(57) Abrégé: L'invention concerne un protocole de gestion et un procédé de vérification d'un fragment de programme, ou appliquette, téléchargé sur un système embarqué. Une commande de téléchargement (100a, 100b) de l'appliquette est effectuée. Sur réponse positive, le code objet de l'appliquette est lu (101) et soumis (102) à une vérification instruction par instruction. La vérification consiste en une étape d'initialisation de la pile des types et du tableau des types de registres représentant l'état de la machine virtuelle du système embarqué au début de l'exécution du code de l'appliquette et en une vérification, instruction par instruction pour chaque instruction courante cible, de l'existence d'une cible d'instruction de branchement, d'appel d'un gestionnaire d'exceptions ou d'un appel de sous-routine, et par une vérification et une actualisation de l'effet de cette instruction sur la pile des types et le tableau des types de registres. Sur vérification réussie (103a), l'appliquette est enregistrée (104) et un accusé de réception est envoyé (105) au lecteur de téléchargement. L'appliquette est détruite (106) sinon. Application aux systèmes embarqués en environnement Java.

PROTOCOL FOR MANAGING AND METHOD OF VERIFYING AND OF

CONVERTING A DOWNLOADED PROGRAM FRAGMENT, AND

CORRESPONDING SYSTEMS

5   The invention relates to a protocol for manag-
ing, a method of verifying and a method of transforming
a downloaded program fragment and the corresponding
systems, more particularly intended for on-board data-
processing systems having few resources available in
10   terms of memory and of computing power.
           In a general way, by reference to figure 1a, it
is reiterated that on-board data-processing systems 10
include a microprocessor 11, a permanent memory, such
as a non-writable memory 12 containing the code of the
15   executable program, and a rewritable, nonvolatile, per-
manent memory 15 of EEPROM type containing the data
stored in the system, a volatile, random-access memory
14 in which the program stores its intermediate results
while it is executing, and input/output devices 15 al-
20   lowing the system to interact with its environment. In
the case in which the on-board data-processing system
consists of a microprocessor card, of the bank-card
type, the input/output device 15 consists of a serial
link allowing the card to communicate with a terminal,
25   such as a card-reader terminal.
           In conventional on-board data-processing sys-
tems, the code of the program executed by the system is
fixed during construction of the system, or, at the
latest, when the latter is customized before delivery
30   to the final user.
           More sophisticated on-board data-processing sys-
tems have, however, been implemented, these systems be-
ing reprogrammable, such as microprocessor cards of the
JavaCard type, for example. With respect to the preced-
35   ing types, these reprogrammable systems add the possi-
bility of enhancing the program after the system has
been put into service, via an operation of downloading
program fragments. These fragments of programs, widely

designated by "applets", will be designated applets or program fragments indiscriminately in the present description. For a more detailed description of JavaCard systems, reference could usefully be made to the documentation published by the company SUN MICROSYSTEMS INC., and in particular to the electronically available documentation, JavaCard technology chapter, on the www (*World Wide Web*) site http://java.sun.com/products/javacard/index.html, June 1999.

figure 1b illustrates the architecture of such a reprogrammable on-board data-processing system. This architecture is similar to that of a conventional on-board system, with the difference that the reprogrammable on-board system can moreover receive applets by way of one of its input/output devices, then store them in its permanent memory 13 from which they can then be executed as a supplement to the main program.

For reasons of portability between different on-board data-processing systems, applets are presented in the form of code for a standard virtual machine. This code is not directly executable by the microprocessor 11, but it has to be interpreted in software terms by a virtual machine 16, which consists of a program resident in a non-writable permanent memory 12. In the abovementioned example of JavaCard cards, the virtual machine used is a subset of the Java virtual machine. For a description of the specifications relating to the Java virtual machine and of the virtual machine used, reference could usefully be made to the work published by Tim LINDHOLM and Frank YELLIN, entitled "The Java Virtual Machine Specification", Addison-Wesley 1996, and to the documentation published by the company SUN MICROSYSTEMS INC. "JavaCard 2.1 Virtual Machine Specification", documentation available electronically on the www site http://java.sun.com/products/javacard/JCVMSpec.pdf, March 1999.

The operation of downloading applets onto an on-
board data-processing system in service poses consider-
able security problems. An applet which is involuntar-
ily, or even deliberately, badly written may incor-
5   rectly modify data present on the system, prevent the
main program from executing correctly or at the right
time, or else modify other applets downloaded previ-
ously, making them unusable or harmful.

An applet written by a computer hacker may also
10   divulge confidential information stored in the system,
information such as the access code in the case of a
bank card, for example. At the present time, three so-
lutions have been proposed with a view to remedying the
problem of security of applets.

15   A first solution consists in using cryptographic
signatures, so as to accept only applets originating
from trusted bodies or persons.

In the abovementioned example of a bank card,
only the applets bearing the cryptographic signature of
20   the bank having issued the card are accepted and exe-
cuted by the card, any other unsigned applet being re-
jected in the course of the downloading operation. An
ill-intentioned user of the card, not having available
encryption keys from the bank, will therefore be inca-
25   pable of executing an unsigned and dangerous applet on
the card.

This first solution is well adapted to the case
where all the applets originate from the same single
source, the bank in the abovementioned example. This
30   solution is difficult to apply in the case in which the
applets originate from several sources, such as, in the
example of a bank card, the manufacturer of the card,
the bank, the bodies managing services by bank card,
the large commercial distribution organizations offer-
35   ing clientele loyalty programs and proposing, legiti-
mately, to download specific applets onto the card. The
sharing and the holding among these various economic
participants of the encryption keys necessary for the

electronic signature of the applets pose major techni-
cal, economic and legal problems.

A second solution consists in carrying out dy-
namic checks on access and on typing during the execu-
5 tion of the applets.

In this solution, the virtual machine carries
out a certain number of checks, during the execution of
the applets, such as:

- check of access to the memory: upon each
10 read or write in a memory area, the virtual machine
verifies the right of access by the applet to the cor-
responding data;

- dynamic verification of the data types: upon
each instruction from the applet, the virtual machine
15 verifies that the constraints on the data types are
satisfied. By way of example, the virtual machine may
have special handling for data such as valid memory ad-
dresses, and prevent the applet generating invalid mem-
ory addresses by way of integer/address conversions or
20 arithmetic operations on the addresses;

- detection of stack overflows and of illegal
accesses to the execution stack of the virtual machine,
which, under certain conditions, are likely to disturb
the operation thereof, to the point of circumventing
25 the preceding check mechanisms.

This second solution allows execution of a wide
range of applets under satisfactory security condi-
tions. However, it features the drawback of a consider-
able slowing of the execution, caused by the range of
30 dynamic verifications. In order to obtain a reduction
in this slowing, some of these verifications can be
taken charge of by the microprocessor itself, at the
cost, however, of an increase in the complexity thereof
and thus of the cost price of the on-board system. Such
35 verifications furthermore increase the requirements for
random-access and permanent memory of the system, by
reason of the additional type information which it is
necessary to associate with the data handled.

A third solution consists in carrying out a static verification of the code of the applet during the downloading.

In this solution, this static verification simu-
5 lates the execution of the applet at the level of the data types and establishes, once and for all, that the code of the applet complies with the rule of data types and of access control imposed by the virtual machine and does not cause a stack overflow. If this static
10 verification is successful, the applet can then be executed without it being necessary dynamically to verify that this rule is complied with. In the event that the static verification process fails, the on-board system rejects the "applet" and does not allow its subsequent
15 execution. For a more detailed description of the abovementioned third solution, reference could usefully be made to the work published by Tim LINDHOLM and Frank YELLIN quoted above, to the article published by James A. GOSLING entitled "Java Intermediate Byte Codes",
20 proceedings of the ACM SIGPLAN, Workshop on Intermediate Representations (IR'95), pages 111-118, January 1995, and to the US patent 5,748,964 granted on 05/05/1998.

Compared with the second solution, the third so-
25 lution presents the advantage of a much more rapid execution of the applets, since the virtual machine does not carry out any verification during execution.

The third solution, however, features the draw-back of a process of static verification of the code
30 which is complex and expensive, both in terms of size of code necessary to conduct this process and in terms of size of random-access memory necessary to contain the intermediate results of the verification, and in terms of computation time. By way of illustrative exam-
35 ple, the code verification integrated into the Java JDK system marketed by SUN MICROSYSTEMS represents about 50 kbytes of machine code, and its consumption in terms of random-access memory is proportional to $(Tp + Tr) \times Nb$,

where Tp designates the maximum stack space, Tr desig-
nates the maximum number of registers and Nb designates
the maximum number of branching targets used by a sub-
program, also widely designated by method, of the ap-
5   plet. These memory requirements greatly exceeded the
capacities of the resources of the majority of the pre-
sent-day on-board data-processing systems, especially
of commercially available microprocessor cards.

     Several variants of the third solution have been
10   proposed, in which the writer of the applet sends to
the verifier, in addition to the code of the applet, a
certain amount of specific supplementary information
such as precalculated data types or preestablished
proof of correct data typing. For a more detailed de-
15   scription of the corresponding operating modes, refer-
ence could usefully be made to the articles published
by Eva ROSE and Kristoffer HØGSBRO ROSE, "Lightweight
Bytecode Verification", proceedings of the Workshop
Formal Underspinning of Java, October 1998, and by
20   George C. NECULA, "Proof-Carrying Code", Proceedings of
the 24th ACM Symposium Principles of Programming Lan-
guages, pages 106-119, respectively.

     This supplementary information makes it possible
to verify the code more rapidly and slightly to reduce
25   the size of the code of the verification program but
does not make it possible, however, to reduce the re-
quirements for random-access memory, or even increases
them, very substantially, in the case of the correct-
data-typing preestablished-proof information.

30   The object of the present invention is to remedy
the abovementioned drawbacks of the prior art.

     In particular, one subject of the present inven-
tion is the implementation of a protocol for managing a
downloaded program fragment, or applet, allowing execu-
35   tion of the latter by an on-board data-processing sys-
tem having few resources available, such as a micro-
processor card.

Another subject of the present invention is also the implementation of a method of verifying a downloaded program fragment, or applet, in which a process of static verification of the code of the applet is

5   conducted when it is downloaded, this process possibly being aligned, at least in its principle, with the third solution described above, but in which new verification techniques are introduced, so as to allow execution of this verification within the limits of values

10   of memory size and of computation speed imposed by the microprocessor cards and other low-power on-board data-processing systems.

Another subject of the present invention is also the implementation of methods of transforming program

15   fragments of conventional type obtained, for example, by the use of a Java compiler on standardized program fragments, or applets, satisfying, a priori, verification criteria of the verification method which is the subject of the invention, with a view to accelerating

20   the process of verifying and executing them at the level of present-day microprocessor cards or on-board data-processing systems.

Another subject of the present invention is, finally, the production of on-board data-processing sys-

25   tems allowing implementation of the abovementioned protocol for managing and of the abovementioned method of verifying a downloaded program fragment as well as of data-processing systems allowing implementation of the methods of transforming conventional program fragments,

30   or applets, into standardized program fragments, or applets, as mentioned above.

The protocol for managing a downloaded program fragment on a reprogrammable on-board system, which is the subject of the present invention, applies espe-

35   cially to a microprocessor card equipped with a rewritable memory. The program fragment consists of an object code, a series of instructions, executable by the microprocessor of the on-board system by way of a vir-

tual machine equipped with an execution stack and with
local variables or registers manipulated via these in-
structions and making it possible to interpret this ob-
ject code. The on-board system is interconnected to a
5   terminal.

It is noteworthy in that it consists at least,
at the level of the on-board system, in detecting a
command for downloading of the program fragment. On a
positive response to the stage consisting in detecting
10   a downloading command, it further consists in reading
the object code constituting the program fragment and
in temporarily storing this object code in the rewri-
table memory. The whole of the object code stored in
memory is subjected to a verification process, instruc-
15   tion by instruction. The verification process consists
at least in a stage of initializing the type stack and
the table of register types representing the state of
the virtual machine at the start of the execution of
the temporarily stored object code and in a succession
20   of stages of verification, instruction by instruction,
of the existence, for each current instruction, of a
target, branching-instruction target, target of an ex-
ception handler, and in a verification and an updating
of the effect of the current instruction on the type
25   stack and on the table of register types. In the event
of an unsuccessful verification of the object code, the
protocol which is the subject of the invention consists
in deleting the momentarily recorded program fragment,
when omitting to record the latter in the directory of
30   available program fragments, and in sending an error
code to the reader.

The method of verifying a program fragment down-
loaded onto an on-board system, which is the subject of
the invention, applies especially to a microprocessor
35   card equipped with a rewritable memory. The program
fragment consists of an object code and includes at
least one subprogram, a series of instructions, execu-
table by the microprocessor of the on-board system by

way of a virtual machine equipped with an execution
stack and with operand registers manipulated by these
instructions, and making it possible to interpret this
object code. The on-board system is interconnected to a
5   reader.

It is noteworthy in that, following the detec-
tion of a downloading command and the storage of the
object code constituting the program fragment in the
rewritable memory, it consists, for each subprogram, in
10   carrying out a stage of initializing the type stack and
the table of register types by data representing the
state of the virtual machine at the start of the execu-
tion of the temporarily stored object code, in carrying
out a verification of the temporarily stored object
15   code instruction by instruction, by discerning the ex-
istence, for each current instruction, of a branching-
instruction target, of a target of an·exception-handler
call or of a target of a subroutine call, and in carry-
ing out a verification and an updating of the effect of
20   the current instruction on the data types of the type
stack and of the table of register types, on the basis
of the existence of a branching-instruction target, of
a target subroutine call or of a target of an excep-
tion-handler call. The verification is successful when
25   the table of register types is not modified in the
course of a verification of all the instructions, the
verification process being carried out instruction by
instruction until the table of register types is sta-
ble, with no modification present. Otherwise the veri-
30   fication process is interrupted.

The method of transforming an object code of a
program fragment into a standardized object code for
this same program fragment, which is the subject of the
present invention, applies to an object code of a pro-
35   gram fragment in which the operands of each instruction
belong to the data types manipulated by this instruc-
tion, the execution stack does not exhibit any overflow
phenomenon and, for each branching instruction, the

type of the variables of the stack at this branching is
the same as at the targets of this branching. The stan-
dardized object code obtained is such that the operands
of each instruction belong to the data types manipu-
5   lated by this instruction, the execution stack does not
exhibit any overflow phenomenon and the execution stack
is empty at each branching-target instruction.

It is noteworthy in that it consists, for all
the instructions of the object code, in annotating each
10   current instruction with the data type of the execution
stack before and after the execution of this instruc-
tion, the annotation data being calculated by means of
an analysis of the data stream relating to this in-
struction, in detecting, within the instructions and
15   within each current instruction, the existence of
branchings for which the execution stack is not empty,
the detection operation being carried out on the basis
of the annotation data of the type of stack variables
allocated to each current instruction. In the presence
20   of a detection of a non-empty execution stack, it fur-
ther consists in inserting instructions to transfer
stack variables on either side of these branchings or
of these branching targets in order to empty the con-
tents of the execution stack into temporary registers
25   before this branching and to reestablish the execution
stack from the temporary registers after this branch-
ing, and in not inserting any transfer instruction oth-
erwise.

This method thus makes it possible to obtain a
30   standardized object code for this same program frag-
ment, in which the execution stack is empty at each
branching instruction and branching-target instruction,
in the absence of any modification to the execution of
the program fragment.

35   The method of transforming an object code of a
program fragment into a standardized object code for
this same program fragment, which is the subject of the
present invention, applies, moreover, to an object code

of a program fragment in which the operands of each in-
struction belong to the data types manipulated by this
instruction, and a operand of given type written into a
register by an instruction of this object code is re-
5   read from this same register by another instruction of
this object code with the same given data type. The
standardized object code obtained is such that the op-
erands belong to the data types manipulated by this in-
struction, one and the same data type being allocated
10 to the same register throughout the standardized object
code.

It is noteworthy in that it consists, for all
the instructions of the object code, in annotating each
current instruction with the data type of the registers
15 before and after the execution of this instruction, the
annotation data being calculated by means of an analy-
sis of the data stream relating to this instruction,
and in carrying out a reallocation of the original reg-
isters employed with different types, by dividing these
20 original registers into separate standardized regis-
ters. One standardized register is allocated to each
data type used. Reupdating of the instructions which
manipulate the operands which use the standardized reg-
isters is carried out.

25 The protocol for managing a program fragment,
the method of verifying a program fragment, the methods
of transforming object code of program fragments into
standardized object code and the corresponding systems,
which are the subjects of the present invention, find
30 an application in the development of reprogrammable on-
board systems, such as microprocessor cards, especially
in the Java environment.

They will be better understood on reading the
description and on perusing the drawings below, in
35 which, other than figures 1a and 1b relating to the
prior art:

      -    Figure 2 represents a flow chart illustrating the protocol for managing a program fragment downloaded onto a reprogrammable on-board system,

5   -   Figure 3a represents, by way of illustration, a flow chart of a method of verifying a downloaded program fragment in accordance with the subject of the present invention,

      -    Figure 3b represents a diagram illustrating data types and sub-typing relationships implemented by

10   the method of managing and the method of verifying a downloaded program fragment, which is the subject of the present invention,

      -    Figure 3c represents a detail of the verification method as claimed in figure 3a, relating to the

15   managing of a branching instruction,

      -    Figure 3d represents a detail of the verification method as claimed in figure 3a, relating to the managing of a subroutine-call instruction,

      -    Figure 3e represents a detail of the verifi-

20   cation method as claimed in figure 3a, relating to the managing of an exception-handler target,

      -    Figure 3f represents a detail of the verification method as claimed in figure 3a, relating to the managing of a target of incompatible branchings,

25   -   Figure 3g represents a detail of the verification method as claimed in figure 3a, relating to the managing of an absence of branching target,

      -    Figure 3h represents a detail of the verification method as claimed in figure 3a, relating to the

30   managing of the effect of the current instruction on the type stack,

      -    Figure 3i represents a detail of the verification method as claimed in figure 3a, relating to the managing of an instruction for reading a register,

35   -   Figure 3j represents a detail of the verification method as claimed in figure 3a, relating to the managing of an instruction for writing to a register,

- Figure 4a represents a flow chart illustrating a method of transforming an object code of a program fragment into a standardized object code for this same program fragment with a branching instruction, re-
5 spectively a branching-target instruction, with an empty stack,

- Figure 4b represents a flow chart illustrating a method of transforming an object code of a program fragment into a standardized object code for this
10 same program fragment, making use of typed registers, a single specific data type being attributed to each register,

- Figure 5a represents a detail of implementation of the transformation method illustrated in figure
15 4a,

- Figure 5b represents a detail of implementation of the transformation method illustrated in figure 4b,

- Figure 6 represents a functional diagram of
20 the complete architecture of a system for development of a standardized program fragment, and of a reprogrammable microprocessor card allowing implementation of the protocol for managing and the method of verifying a program fragment in accordance with the subject of the
25 present invention.

In general, it is indicated that the protocol for managing and the method of verifying and transforming a downloaded program fragment, which is the subject of the present invention, and the corresponding sys-
30 tems, are implemented thanks to a software architecture for secure downloading and execution of applets on an on-board data-processing system with few resources, such as, in particular, microprocessor cards.

In general, it is indicated that the description
35 below concerns the application of the invention in the context of reprogrammable microprocessor cards of JavaCard type, cf. documentation available electronically from the company SUN MICROSYSTEMS INC., JavaCard

Technology heading mentioned previously in the descrip-
tion.

However, the present invention is applicable to
any on-board system which is reprogrammable by down-
loading an applet which is written in the code of a
virtual machine including an execution stack, local
variables or registers, and of which the execution
model is strongly typed, each instruction of the code
of the applet applying only to specific data types. The
protocol for managing a program fragment downloaded
onto a reprogrammable on-board system, which is the
subject of the present invention, will now be described
in more detail with reference to fig. 2.

With reference to the abovementioned figure, it
is indicated that the object code which makes up the
program fragment or applet consists of a series of in-
structions which can be executed by the microprocessor
of the on-board system by means of the abovementioned
virtual machine. The virtual machine makes it possible
to interpret the abovementioned object code.
The on-board system is interconnected to a terminal
via, for instance, a serial link.

With reference to the abovementioned fig. 2, the
management protocol which is the subject of the present
invention consists at least, in the on-board system,
in detecting a command to download this program frag-
ment in a stage 100a, 100b. Thus, stage 100a may con-
sist of a stage of reading the abovementioned command,
and stage 100b of a stage of testing the command which
has been read and verifying the existence of a down-
loading command.

On a positive response to the abovementioned
stage 100a, 100b of detecting a downloading command,
the protocol which is the subject of the present inven-
tion subsequently consists in reading, at stage 101,
the object code which makes up the relevant program
fragment, and temporarily storing the abovementioned
object code in the memory of the on-board data-

processing system. The abovementioned temporary storing
operation can be executed either in the rewritable mem-
ory or, if appropriate, in the random-access memory of
the on-board system, when this has sufficient capacity.

5   The stage of reading the object code and temporarily
storing it in the rewritable memory is designated as
loading the code of the applet in fig. 2.

        The abovementioned stage is then followed by a
stage 102 consisting in submitting the whole of the
10 temporarily stored object code to a process of verifi-
cation, instruction by instruction, of the abovemen-
tioned object code.

        The verification process consists, at least in a
stage of initializing the stack of types and the table
15 of data types representing the state of the virtual ma-
chine at the start of execution of the temporarily
stored object code, and in a succession of stages of
verifiying, instruction by instruction, by discerning
the existence, for each current instruction, designated
20 I_i, of a target such as a branching-instruction target
designated CIB, a target of an exception-handler call
or a target of a subroutine call. A verification and
update of the effect of the current instruction I_i on
the stack of types and on the table of register types
25 is carried out.

        When the verification has been successful at
stage 103a, the protocol which is the subject of the
present invention consists in recording, at stage 104,
the downloaded program fragment in a directory of
30 available program fragments, and in sending to the
reader, at stage 105, a positive reception acknowledg-
ment.

        On the other hand, in the case of unsuccessful
verification of the object code at stage 103b, the pro-
35 tocol which is the subject of the present invention
consists in inhibiting, in a stage 103c, any execution
on the on-board system of the momentarily recorded pro-
gram fragment. The inhibition stage 103c can be imple-

mented in various ways. As a nonlimiting example, this
stage can consist in deleting, at stage 106, the momen-
tarily recorded program fragment, without recording
this program fragment in the directory of available

5   program fragments and, at stage 107, in sending an er-
ror code to the reader. Stages 107 and 105 can be im-
plemented either sequentially after stages 106 and 104
respectively, or in multitasking operation with them.

      With reference to the same fig. 2, on a negative

10   response to the stage consisting in detecting a down-
loading command at stage 100b, the protocol which is
the subject of the present invention consists in de-
tecting, in a stage 108, a command to select an avail-
able program fragment from a directory of program frag-

15   ments and, on a positive response to stage 108, having
detected the selection of an available program frag-
ment, in calling, at stage 109, this selected available
program fragment in order to execute it. Stage 109 is
then followed by a stage 110 of executing the called

20   available program fragment by way of the virtual ma-
chine, with no dynamic verification of variable types,
rights of access to the objects which are manipulated
by the called available program fragment, or overflow
of the execution stack when each instruction is exe-

25   cuted.

      In the case where a negative response is ob-
tained at stage 108, this stage consisting in detecting
a command to select a called available program frag-
ment, the protocol which is the subject of the present

30   invention consists in proceeding, in a stage 111, to
process the standard commands of the on-board system.

      Regarding the absence of dynamic verification of
type or rights of access to objects of, for instance,
JavaCard type, it is indicated that this absence of

35   verification does not compromise the security of the
card, because the code of the applet has necessarily
successfully passed verification.

More specifically, it is indicated that the code
verification which is carried out, as claimed in the
method which is the subject of the present invention,
on the microprocessor card or on-board data-processing

5   system is more selective than the customary verifica-
tion of codes for the virtual Java machine as described
in the work entitled "The Java Virtual Machine Specifi-
cation" which was mentioned previously in the descrip-
tion.

10   However, any code of the Java virtual machine
which is correct as far as the traditional Java veri-
fier is concerned can be transformed into an equivalent
code which is capable of passing successfully the code
verification which is carried out on the microprocessor

15   card.

Whereas it is possible to imagine writing di-
rectly Java codes which satisfy the abovementioned
verification criteria mentioned in the context of im-
plementing the protocol which is the subject of the

20   present invention, a noteworthy object of the latter is
also the implementation of a method of automatic trans-
.formation of any standard Java code into a standardized
code for the same program fragment, necessarily satis-
fying the verification criteria implemented as men-

25   tioned above. The method of transformation into stan-
dardized code, and the corresponding system, will be
described in detail subsequently in the description.

A more detailed description of the method of
verifying a program fragment, or applet, in accordance

30   with the subject of the present invention, will now be
given with reference to fig. 3a and the subsequent fig-
ures.

In general, it is indicated that the verifica-
tion method which is the subject of the present inven-

35   tion can be implemented either as part of the protocol
for managing a program fragment which is the subject of
the present invention as described above with reference

to fig. 2, or independently, to provide whatever veri-
fication process is judged necessary.

   In general, it is indicated that a program frag-
ment is made up of an object code including at least
5 one subprogram, more commonly designated a method, and
is made up of a series of instructions which can be
executed by the microprocessor of the on-board system
via the virtual machine.

   As shown in fig. 3a, the verification method
10 consists, for each subprogram, in carrying out a stage
200 of initializing the stack of types and the table of
register types of the virtual machine by data repre-
senting the state of this virtual machine at the start
of execution of the object code which is the subject of
15 verification. This object code can be stored temporar-
ily as described above with reference to implementation
of the protocol which is the subject of the present in-
vention.

   The abovementioned stage 200 is then followed by
20 a stage 200a consisting in positioning the reading of
the current instruction I_i, index i, on the first in-
struction of the object code. Stage 200a is followed by
a stage 201 consisting in carrying out a verification
of the abovementioned object code, instruction by in-
25 struction, by discerning the existence, for each cur-
rent instruction, designated I_i, of a branching-
instruction target CIB, of a target of an exception-
handler call, designated CEM, or of a target of a sub-
routine call CSR.

30   The verification stage 201 is followed by a
stage 202 of verifying and updating the effect of the
current instruction I_i on the data types of the stack of
types and of the table of register types, as a function
of the existence, for the current instruction which is
35 pointed at by another instruction, of a branching-
instruction target CIB, of a target of a subroutine
call CSR or a target of an exception-handler call CEM.

Stage 202 for the current instruction I_i is fol-
lowed by a stage 203 to test whether the last instruc-
tion has been reached, the test written as:

I_i = last instruction of the object code?

5   On a negative response to test 203, the process passes
to the next instruction 204, written i = i+1, and to
the return to stage 201.

It is indicated that the abovementioned verifi-
cation, at stage 202, has been successful when the ta-
10   ble of register types is not modified during verifica-
tion of all the instructions I_i which make up the object
code. For this purpose, a test 205 of the existence of
a stable state of the table of register types and pro-
vided. This test is written:

15   ∃? Stable state of table of register types.

On a positive response to test 205, verification has
been successful.

On the other hand, in the case where no absence
of modification is noticed, the verification process is
20   repeated and reinitiated by returning to stage 200a. It
is demonstrated that the process is guaranteed to end
after a maximum of NrxH iterations, where Nr designates
the number of registers used and H designates a con-
stant depending on the subtyping relation.

25   Various   indications   concerning   the   types   of
variables which are manipulated in the course of the
verification process described above with reference to
fig. 3a will now be given with reference to fig. 3b.

The   abovementioned   variable   types   include   at
30   least class identifiers corresponding to object classes
which are defined in the program fragment which is sub-
jected to verification, numeric variable types includ-
ing at least a type short, an integer coded on p bits,
where the value of p can be 16, and a type for the re-
35   turn address of a jump instruction JSR, this address
type being identified as retaddr, a type null relating
to references of null objects, a type object relating
to objects proper, a specific type ⊥ representing the

intersection of all the types and corresponding to the
value zero nul, another specific type $\underline{T}$ representing
the union of all the types and corresponding to any
type of values.

5   With reference to fig. 3b, it is indicated that
all the abovementioned variable types verify a subtyp-
ing relation:

          object ε T;

          short, retaddr ε T;

10   ⊥ ε null, short, retaddr

          A more specific example of a process of verifi-
cation as illustrated in fig. 3a will now be given,
with reference to a first example of a data structure,
which is shown in table T1 in the annex.

15   The abovementioned example concerns an applet
written in Java code.

          The verification process accesses the code of
the subprogram which forms the applet which is sub-
jected to verification via a pointer to instruction I_i

20 which is being verified.

          The verification process records the size and
type of the execution stack at the current instruction
I_i corresponding to saload in the example of the above-
mentioned Table T1.

25   The verification process then records the size
and type of the execution stack at the current instruc-
tion in the stack of types via its type stack pointer.

          As mentioned above in the description, this
stack of types reflects the state of the execution

30 stack of the virtual machine at the current instruction
I_i. In the example shown in table T1, at the time of the
future execution of instruction I_i, the stack will con-
tain three entries: a reference to an object of class
C, a reference to a table of integers coded on p = 16

35 bits, the type short[], and an integer of p = 16 bits
of type short. This is also shown in the type stack,
which also contains three entries: C, the type of ob-

jects of class C, short[], the type of tables of inte-
gers p = 16 bits and short, the type of integers p = 16
bits.

5   Another noteworthy data structure consists of a
table of register types, this table reflecting the
state of the registers, that is to say of the registers
which store the local variables, of the virtual ma-
chine.

10   Continuing the example indicated in table T1, it
is indicated that entry 0 of the table of register
types contains type C, i.e. at the time of the future
execution of the current instruction I_i = saload, regis-
ter 0 is guaranteed to contain a reference to an object
of class C.

15   The various types which are manipulated during
verification and stored in the table of register types
and in the type stack are represented in fig. 3b. These
types include:
-       class identifiers CB corresponding to specific
20   object classes which are defined in the applet;
-       base types, such as short, an integer coded on
        p = 16 bits, int1 and int2, the most and least
        significant p bits respectively of integers
        coded on, e.g., 2p bits, or retaddr, the return
25   address of an instruction as mentioned above;
-       the type null, representing the references of
        null objects.

        Regarding the subtyping relation, it is indi-
cated that a type T1 is a subtype of a type T2 if every
30   valid value of type T1 is also a valid value of type
T2. The subtyping between class identifier reflects the
inheritance hierarchy between classes of the applet. On
the other types, subtyping is defined by the lattice
shown in fig. 3b, where $\perp$ is a subtype of all types and
35   all types are subtypes of $\top$.

The sequence of the process of verifying a sub-
program which forms an applet is as follows, referring
to the abovementioned table T1.

The verification process is carried out inde-
5   pendently on each subprogram of the applet. For each
subprogram, the process carries out one or more verifi-
cation passes on the instructions of the relevant sub-
program. The pseudocode of the verification process is
given in table T2 in the annex.

10   The process of verifying a subprogram begins
with initializing the type stack and the table of reg-
ister types shown in table T1, this initialization re-
flecting the state of the virtual machine at the start
of execution of the subprogram being examined.

15   The type stack is initially empty, the stack
pointer equals zero, and the register types are ini-
tialized with the types of the parameters of the sub-
program, illustrating the fact that the virtual machine
passes the parameters of this subprogram in these reg-
20   isters. The register types which are allocated by the
subprogram are initialized to data types $\perp$, illustrat-
ing the fact that the virtual machine initializes these
registers to zero at the start of execution of the sub-
program.

25   Next, one or more verification passes on the in-
structions and on each current instruction I_i of the
subprogram are carried out.

At the end of the implemented verification pass,
or of a succession of passes for instance, the verifi-
30   cation process determines whether the register types
contained in the table of register types shown in table
T1 of the annex have changed during the verification
pass. In the absence of change, verification is termi-
nated and a success code is returned to the main pro-
35   gram, which makes it possible to send the positive re-
ception acknowledgment at stage 105 of the management
protocol shown in fig. 2.

If a change to the abovementioned table of register types is present, the verification process repeats the verification pass until the register types contained in the table of register types is stable.

5   The sequence proper of a verification pass which is carried out one or more times until the table of register types is stable will now be described with reference to figs. 3c to 3j.

For each current instruction I_i, the following 10 verifications are carried out:

With reference to fig. 3a at stage 201, the verification process determines whether the current instruction I_i is the target of a branching instruction, a subroutine call or an exception-handler call, as mentioned above. This verification is carried out by examining the branching instructions in the code of the subprogram and the exception handlers associated with this subprogram.

With reference to fig. 3c which begins with 20 stage 201, when the current instruction I_i is the target of a branching instruction, this condition being implemented by a test 300 designated by I_i = CIB, this branching being unconditional or conditional, the verification process checks that the type stack is empty at this point of the subprogram by a test 301. On a positive response to the test 301, the verification process is continued by a context continuation stage marked continue A. On a negative response to the test 301, the type stack not being empty, the verification fails and 30 the applet is rejected. This failure is represented by the Failure stage.

With reference to fig. 3d which begins with the continue A stage, when the current instruction I_i is the target of a subroutine call, this condition being implemented by a test 304 I_i = CSR, the verification process verifies, in a test 305, that the previous instruction I_{i-1} does not continue in sequence. This verification is implemented by a test stage 305 when the previ-

ous instruction is an unconditional branching, a sub-
routine return or a raising of an exception. The test
at stage 305 is marked as follows:

$I_{i-1} = IB_{unconditional}$, return RSR or raising L-
5   EXCEPT.

On a negative response to test 305, the verifi-
cation process fails in a Failure stage. On the other
hand, on a positive response to test 305, the verifica-
tion process reinitializes the type stack in such a way
10   that it contains exactly one entry of _retaddr_ type, the
return address of the abovementioned subroutine. If the
current instruction I_i at stage 304 is not the target of
a subroutine call, the verification process is contin-
ued in the context at the continue B stage.

15   With reference to fig. 3e, when the current in-
struction I_i is the target of an exception handler, this
condition being implemented by a test 307 marked I_i =
CEM, where CEM designates the target of an exception
handler, this condition is implemented by a test 307,
20   marked:

$I_i = CEM$.

On a positive response to test 307, the process veri-
fies that the previous instruction is an unconditional
branching, a subroutine return or a raising of excep-
25   tions by a test 305, marked:

$I_{i-1} = IB_{unconditional}$, return RSR or raising L-
EXCEPT.

On a positive response to test 305, the verifi-
cation process proceeds to reupdate the type stack, at
30   a stage 308, by entering exception types, marked EXCEPT
type, stage 308 being followed by a context continua-
tion stage, continue C. On a negative response to test
305, the verification process fails by the stage marked
Failure. The program fragment is then rejected.

35   With reference to fig. 3f, when the current in-
struction I_i is the target of multiple incompatible
branchings, this condition is implemented by a test
309, which is marked:

I_i = incompatible XIBs

incompatible branchings being, for instance, an uncon-
ditional branching and a subroutine call, or two dif-
ferent exception handlers. On a positive response to
5   test 309, the branchings being incompatible, the veri-
fication process fails by a stage marked Failure and
the program fragment is rejected. On a negative re-
sponse to test 309, the verification process is contin-
ued by a stage marked continue D. Test 309 begins with
10   the continue C stage which was mentioned previously in
the description.

With reference to fig. 3g, when the current in-
struction I_i is not the target of any branching, this
condition being implemented by a test 310 beginning
15   with the abovementioned continue D, this test being
marked

I_i ∃? branching targets,

where ∃ designates the existence symbol,

the verification process continues on a negative re-
20   sponse to the test 310 by passing to an update of the
type stack at stage 311, stage 311 and the positive re-
sponse to test 310 being followed by a context con-
tinuation stage at stage 202, which is described above
in the description with reference to fig. 3a.

25   A more detailed description of the stage of
verifying the effect of the current instruction on the
type stack at the abovementioned stage 202 will now be
given with reference to fig. 3h.

According to the abovementioned figure, this
30   stage can include at least one stage 400 of verifica-
tion that the type execution stack includes at least as
many entries as the current instruction includes oper-
ands. This test stage 400 is marked:

Nbep $\geq$ NOpi

35   where Nbep designates the number of entries of the type
stack and NOpi designates the number of operands in-
cluded in the current instruction.

On a positive response to test 400, this test is followed by a stage 401a of unstacking the type stack, and of verifying 401b that the types of the entries at the top of the stack are subtypes of the types of the

5   operands of the abovementioned current instruction. At test stage 401a, the operand types of the instruction i are marked TOpi, and the types of the entries at the top of the stack are marked Targs.

At stage 401b, the verification corresponds to a

10   verification of the subtyping relation Targs subtype of TOpi.

On a negative response to tests 400 and 401b, the verification process fails, which is shown by access to the Failure stage. On the other hand, on a

15   positive response to test 401b, the verification process is continued, and consists in carrying out:

- A stage of verifying the existence of a sufficient memory space on the type stack to proceed to stack the results of the current instruction. This

20   verification stage is implemented by a test 402, marked:

Stack-space $\geq$ Results-space

where each side of the inequality designates the corresponding memory space.

25   On a negative response to test 402, the verification process fails, which is shown by the Failure stage. On the other hand, on a positive response to test 402, the verification process then proceeds to stack the data types which are assigned to the results

30   in a stage 403, the stacking being done on the stack of data types which are assigned to these results.

As a nonlimiting example, it is indicated that to implement fig. 3h for verifying the effect of the current instruction on the type stack, for a current

35   instruction consisting of a Java saload instruction corresponding to reading an integer element coded on p = 16 bits in a table of integers, this table of integers being defined by the table of integers and an in-

teger index in this table, and the result by the inte-
ger which is read at this index in this table, the
verification process checks that the type stack con-
tains at least two elements, that the two elements at
5   the top of the type stack are subtypes of short[] and
short respectively, proceeds to the unstacking process
and then to the process of stacking the data type short
as the result type.

Additionally, with reference to fig. 3i, to im-
10   plement the stage of verifying the effect of the cur-
rent instruction on the type stack, when the current
instruction I_i is a read instruction, marked IR, of a
register of address n, this condition being implemented
by a test 404 marked $I_i = IR_n$, on a positive response to
15   the abovementioned test 404, the verification process
consists in verifying the data type of the result of
this reading, in a stage 405, by reading the entry n in
the table of register types, then in determining the
effect of the current instruction I_i on the type stack
20   by an operation 406a of unstacking the entries of the
stack corresponding to the operands of this current in-
struction and by stacking 406b the data type of this
result. The operands of the instruction I_i are marked
OP_i. Stages 406a and 406b are followed by a return to
25   the context continuation, continue F. On a negative re-
sponse to test 404, the verification process is contin-
ued by the context continuation, continue F.

With reference to fig. 3j, when the current in-
struction I_i is a write instruction, marked IW, of a
30   register of address n, this condition being implemented
by a test marked $I_i = IW_m$, on a positive response to
test 407, the verification process consists in deter-
mining, in a stage 408, the effect of the current in-
struction on the type stack and the type t of the oper-
35   and which is written in the register of address n,
then, in a stage 409, in replacing the type entry of
the table of register types at address n by the type
immediately above the previously stored type and above

the type t of the operand which is written in the reg-
ister of address n. Stage 409 is followed by a return
to the context continuation, continue 204. On a nega-
tive response to test 407, the verification process is
5   continued by a context continuation, continue 204.

As an example, when the current instruction I_i
corresponds to writing a value of type D into a regis-
ter of address 1, and the type of register 1 before
verification of the instruction was C, the type of reg-
10   ister 1 is replaced by the type object, which is the
smallest type which is higher than C and D in the lat-
tice of types shown in fig. 3b.

In the same way, as an example, when the current
instruction I_i is a read of an instruction aload-0 con-
15   sisting in stacking the contents of register 0, and en-
try 0 of the table of register types is C, the verifier
stacks C onto the type stack.

An example of verifying a subprogram written in
a Java environment will now be given, with reference to
20   tables T3 and T4 in the annex.

Table T3 represents a specific JavaCard code
corresponding to the Java subprogram which is included
in this table.

table T4 shows the contents of the table of reg-
25   ister types and of the type stack before verification
of each instruction. The type constraints on the oper-
ands of the various instructions are all observed. The
stack is empty both after the instruction 5 to branch
to instruction 9, symbolized by the arrow, and before
30   the abovementioned branching target 9. The type of reg-
ister 1, which was initially $\perp$, becomes null, the upper
bound of null and $\perp$, when instruction 1 to store a
value of type null in register 1 is examined, then be-
comes of type short[], the upper bound of types short[]
35   and null, when instruction 8 to store a value of type
short[] in register 1 is processed. The type of regis-
ter 1 having changed during the first verification

pass, a second pass is carried out, distributing regis-
ter types which were obtained at the end of the first.
This second verification pass is successful, just like
the first, and does not change the register types. The
5   verification process thus terminates successfully.

Various examples of cases of failure of the
verification process on four examples of incorrect code
will now be given with reference to table T5 in the an-
nex:

10   -   At point a) of table T5, the purpose of the code
           given as an example is to attempt to fabricate
           an invalid object reference using an arithmetic
           process on pointers. It is rejected by verifica-
           tion of the types of arguments of instruction 2
15   sadd, which requires these two arguments to be
           of type short.
      -    At points b) and c) of table T5, the purpose of
           the code is to carry out two attempts to trans-
           form any integer into an object reference. At
20   point b), register 0 is used simultaneously with
           type short, instruction 0, and with type null,
           instruction 5. Consequently, the verification
           process assigns type T to record 0, and detects
           a type error when register 0 is returned as a
25   result of type object at instruction 7.
      -    At point c) of table T5, a set of branchings of
           type "if ... then ... else ..." is used to leave
           at the top of the stack a result which consists
           of either an integer or an object reference. The
30   verification process rejects this code because
           it detects that the stack is not empty at the
           branching from instruction 5 to instruction 9,
           symbolized by the arrow.
      -    Finally, at point d) of table 5, the code con-
35   tains a loop which, at each iteration, has the
           effect of stacking an additional integer on the
           top of the stack, and thus causing a stack over-
           flow after a certain number of iterations. The

verification process rejects this code, noticing
that the stack is not empty at the backward
branching from instruction 8 to instruction 0,
symbolized by the return arrow, the stack not
5   being empty at a branching point.

The various examples given above with reference
to tables T3, T4 and T5 show that the verification
process, which is the subject of the present invention,
is particularly efficient, and that it applies to ap-
10   plets, and in particular to subprograms thereof, for
which the conditions of stack type, or respectively of
the empty character of the type stack previously, and
to the branching instructions or branching targets, are
satisfied.

15   Obviously, such a verification process implies
writing object codes which satisfy these criteria,
these object codes possibly corresponding to the sub-
program in the abovementioned table T3.

However, and in order to ensure verification of
20   existing applets and subprograms of applets which do
not necessarily satisfy the verification criteria of
the method which is the subject of the present inven-
tion, in particular regarding applets and subprograms
which are written in the Java environment, the purpose
25   of the present invention is to establish methods of
transforming these applets or subprograms into stan-
dardized applets or program fragments, making it possi-
ble to undergo successfully the verification tests of
the verification method which is the subject of the
30   present invention and of the management protocol which
implements such a method.

For this purpose, the subject of the invention
is the implementation of a method and a program for
transforming a traditional object code forming an ap-
35   plet, it being possible to implement this method and
this transformation program, outside an on-board system
or microprocessor card, when the relevant applet is
created.

The method of transforming code into standard-
ized code, which is the subject of the present inven-
tion, will now be described in the framework of the
Java environment, as a purely illustrative example.

5   The JVM codes which are produced by existing
Java compilers satisfy various criteria, which are
stated below:

C1:   the arguments of each instruction actually be-
       long to the types which this instruction ex-
10   pects;

C2:   the stack does not overflow;

C'3:   for each branching instruction, the type of the
       stack at this branching is the same as at the
       possible targets for this branching;

15 C'4:   a value of type t which is written into a regis-
       ter at one point of the code and reread from the
       same register at another point of the code is
       always reread with the same type t;

       The implementation of the verification method
20 which is the subject of the present invention implies
that criteria C'3 and C'4, which are verified by the
object code which is submitted for verification, be re-
placed by criteria C3 and C4 below:

C3:   the stack is empty at each branching instruction
25   and at each branching target;

C4:   the same register is used with one and the same
       type throughout the code of a subprogram.

       With reference to the abovementioned criteria,
it is indicated that Java compilers guarantee only the
30 weaker criteria C'3 and C'4. The verification process
which is the subject of the present invention and the
corresponding management protocol in fact guarantee
more restrictive criteria C3 and C4, making it possible
to ensure the security of execution and management of
35 applets.

       The concept of standardization, covering the
transformation of codes into standardized codes, can
present various aspects, to the extent that replacement

of criteria C'3 and C'4 by criteria C3 and C4, in con-
formity with the verification process which is the sub-
ject of the present invention, can be implemented inde-
pendently, to ensure that the stack is empty at each
5 branching instruction and at each branching target, and
respectively that the registers which the applet opens
are typed, and a single data type which is assigned for
execution of the relevant applet corresponds to each
open register, or, on the other hand, jointly, to sat-
10 isfy the whole of the verification process which is the
subject of the present invention.

The method of transforming an object code into
standardized object code as claimed in the invention
will consequently be described as claimed in two dis-
15 tinct implementation modes, a first implementation mode
corresponding to transformation of an object code which
satisfies criteria C1, C2, C'3, C'4 into a standardized
object code which satisfies criteria C1, C2, C3, C'4
corresponding to a standardized code with an empty
20 branching instruction or branching target, then, as
claimed in a second implementation mode, in which the
traditional object code which satisfies the same ini-
tial criteria is transformed into a standardized object
code which satisfies criteria C1, C2, C'3, C4, for in-
25 stance corresponding to a standardized code using typed
registers.

The first implementation mode of the code trans-
formation method which is the subject of the present
invention will now be described with reference to fig.
30 4a. In the implementation mode which is shown in fig.
4a, the initial traditional code is considered to sat-
isfy criteria C1+C2+C'3, and the standardized code
which is obtained as the result of the transformation
is considered to satisfy criteria C1+C2+C3.

35 According to the abovementioned figure, the
transformation method consists, for each current in-
struction I_i of the code or of the subprogram, in anno-
tating each instruction, in a stage 500, with the data

type of the stack before and after execution of this
instruction. The annotation data is marked AI_i and is
associated by the relation $I_i \Leftrightarrow AI_i$ with the relevant
current instruction. The annotation data is calculated
by means of an analysis of the data stream relating to
this instruction. The data types before and after exe-
cution of the instruction are marked tbe_i and tae_i re-
spectively. Calculation of annotation data by analysis
of the data stream is a traditional calculation which
is known to those skilled in the art, and will there-
fore not be described in detail.

The operation which is implemented at stage 500
is illustrated in table T6 in the annex, in which, for
an applet or subprogram of an applet including 12 in-
structions, the annotation data AI_i made up of the types
of registers and the types of the stack are introduced.

The abovementioned stage 500 is then followed by
a stage 500a consisting in positioning the index i on
the first instruction $I_i = I_1$. Stage 500a is followed by
a stage 501 consisting in detecting, among the instruc-
tions and in each current instruction I_i, the existence
of branchings marked IB or of branching targets CIB for
which the execution stack is not empty. This detection
501 is implemented by a test which is carried out on
the basis of the annotation data AI_i of the type of
stack variables allocated to each current instruction,
the test being marked for the current instruction:

I_i is an IB or CIB and stack (AI) ? empty.

On a positive response to test 501, i.e. in the
presence of detection of a non-empty execution stack,
the abovementioned test is followed by a stage consist-
ing in inserting instructions to transfer stack vari-
ables on either side of these branchings IB or branch-
ing targets CIB, in order to empty the content of the
execution stack into temporary registers before this
branching and to reestablish the execution stack from
the temporary registers after this branching. The in-
sertion stage is marked 502 in fig. 4a. It is followed

by a stage 503 to test reaching the last instruction,
marked

I_i = last instruction?

On a negative response to the test 503, an increment
504 i=i+1 is carried out, to go on to the next instruc-
tion and return to stage 501. On a positive response to
test 503, an End stage is initiated. On a negative re-
sponse to test 501, the transformation method is con-
tinued by a branching to stage 503 in the absence of
insertion of a transfer instruction. Implementation of
the method of transforming a traditional code into a
standardized code with branching instruction with empty
stack as represented in fig. 4a makes it possible to
obtain a standardized object code for the same initial
program fragment in which the stack of stack variables
is empty at each branching instruction and each branch-
ing-target instruction, in the absence of any modifica-
tion to the execution of the program fragment. In the
case of a Java environment, the instructions to trans-
fer data between stack and register are the load and
store instructions of the Java virtual machine.

Returning to the example introduced in table T6,
the transformation method detects a branching target
where the stack is not empty at instruction 9. The
method is then to insert an instruction istore 1 before
the branching instruction 5 which leads to the above-
mentioned instruction 9, in order to save the content
of the stack in register 1 and ensure that the stack is
empty at the time of the branching. Symmetrically, an
instruction iload 1 is inserted before the instruction
target 9, to reestablish the content of the stack ex-
actly as it was before the branching. Finally, an in-
struction istore 1 is inserted after instruction 8 to
ensure that the stack is balanced on the two paths
which lead to instruction 9. The result of the trans-
formation carried out in this way into a standardized
code is shown in table T7.

The second implementation mode of the transfor-
mation method which is the subject of the present in-
vention will now be described with reference to fig. 4b
in the case in which the initial traditional object
code satisfies criteria C1+C'4 and the standardized ob-
ject code satisfies criteria C1+C4.

With reference to the abovementioned fig. 4b, it
is indicated that the method, in this implementation
mode, consists in annotating, as claimed in a stage 500
which is approximately the same as that which is shown
in fig. 4a, each current instruction I_i with the type of
register data before and after execution of this in-
struction. In the same way, the annotation data AI_i is
calculated by means of an analysis of the data stream
relating to this instruction.

The annotation stage 500 is then followed by a
stage consisting in carrying out a reallocation of the
registers, the stage marked 601, by detecting the
original registers employed with different types, by
dividing these original registers into separate stan-
dardized registers, one standardized register being al-
located to each data type used. Stage 601 is followed
by a stage 602 of reupdating the instructions which ma-
nipulate the operands which use the abovementioned
standardized registers. Stage 602 is followed by a con-
text continuation stage 302.

With reference to the example given in table T6,
it is indicated that the transformation method detects
that the register of rank 0, marked r0, is used with
the two types, _object_, instructions 0 and 1, and _int_,
instruction 9 and following. The method is then to di-
vide the original register r0 into two registers, reg-
ister 0 for the use of _object_ types and register 1 for
uses of _int_ type. References to record 0 of _int_ type
are then rewritten by transforming them into references
to record 1, the standardized code obtained being shown
in table T8 in the annex.

It is noted, in a nonlimiting way, that in the example which is introduced with reference to the abovementioned table T8, the new register 1 is used simultaneously for standardization of the stack and for creation of typed registers by dividing of register 0 into two registers.

The method of transforming a traditional code into a standardized code with branching instruction with empty stack as described in fig. 4a will now be described in more detail in a preferred, nonlimiting implementation mode, in relation to fig. 5a.

This implementation mode concerns stage 501, consisting in detecting, within the instructions and within each current instruction I_i, the existence of branching IB, or respectively of branching target CIB, for which the stack is not empty.

Following the determination of target instructions where the stack is not empty, this condition being marked at stage 504a, I_i stack$\neq$empty, the transformation process consists in associating with these instructions, at the abovementioned stage 504a, a set of new registers, one per stack location which is active at these instructions. Thus, if i designates the rank of a branching target of which the associated stack type is not empty and is of type $tp1_i$ to tpn_i with $n > 0$, stack not empty, the transformation process allocates n new, as yet unused, registers, r_1 to r_n, and associates them with the corresponding instruction i. This operation is implemented at stage 504a.

Stage 504a is followed by a stage 504 consisting in examining each detected instruction of rank i and identifying, in a test stage 504, the existence of a branching target CIB or of a branching IB. Stage 504 is shown in the form of a test designated by:

I_i?CIB,IB and I_i=CIB.

In the case that the instruction of rank i is a branching target CIB represented by the preceding

equality, and that the stack of stack variables at this
instruction is not empty, i.e. with a positive response
to test 504, for every preceding instruction of rank
i-1 consisting of a branching, a raising of an excep-
5   tion or a program return, this condition is implemented
at test stage 505, designated by:

$I_{i-1} = IB$, EXCEPT raising, Prog. return.

The detected instruction of rank i is only accessible
by a branching. On a positive response to the abovemen-
10   tioned test 505, the transformation process consists in
carrying out a stage 506 consisting in inserting a set
of load instructions of _load_ type from the set of new
registers before the relevant detected instruction of
rank i. The insertion operation 506 is followed by a
15   redirection 507 of all branchings to the detected in-
struction of rank i, to the first inserted load in-
struction _load_. The insertion and redirection opera-
tions are shown in table T9 in the annex.

For every preceding instruction of rank i-1 con-
20   tinuing in sequence, i.e. when the current instruction
of rank i is accessible simultaneously by a branching
and from the preceding instruction, this condition be-
ing implemented by test 508 and symbolized by the rela-
tions:

25       $I_{i-1} \rightarrow I_i$

and

         $IB \rightarrow I_i$

the transformation process consists in a stage 509 of
inserting a set of backup instructions _store_ to back up
30   to the set of new registers before the detected in-
struction of rank i, and a set of load instructions
load to load from this set of new registers. Stage 509
is then followed by a stage 510 of redirection of all
the branchings to the detected instruction of rank i to
35   the first inserted load instruction _load_.

In the case that the detected instruction of
rank i is a branching to a determined instruction, for

any detected instruction of rank i consisting of an un-
conditional branching, this condition being implemented
by a test 511 marked:

   $I_i=IB_{uncondit.}$

5   the transformation process as shown in fig. 5a consists
in inserting at a stage 512, on a positive response to
test 511, before the detected instruction of rank i,
multiple backup instructions store. The transformation
process inserts before the instruction i the n instruc-

10   tions store as shown in table T11 as an example. The
instructions store address registers r_1 to r_n, where n
designates the number of registers. Thus the backup in-
struction is associated with each new register.

      For every detected instruction of rank i con-
15   sisting of a conditional branching, and for a number
mOp, greater than 0, of operands manipulated by this-
conditional branching instruction, this condition being
implemented by the test 513 marked:

   $I_i=IB_{condit.}$

20   with mOp > 0
the transformation process, in positive response to the
abovementioned test 513, consists of inserting, at a
stage 514 before this detected instruction of rank i, a
permutation instruction marked swap_x at the top of the

25   stack of stack variables of the mOp operands of the de-
tected instruction of rank i and the n following val-
ues. This permutation operation makes it possible to
collect at the top of the stack of stack variables the
n values to be backed up in the set of new registers r_1

30   to r_n. Stage 514 is followed by a stage 515 consisting
in inserting, before the instruction of rank i, a set
of backup instructions store to back up to the set of
new registers r_1 to r_n. The abovementioned insertion
stage 515 is itself followed by a stage 516 of inser-

35   tion, after the detected instruction of rank i, of a
set of load instructions load to load from the set of
new registers r_1 to r_n. The set of corresponding inser-
tion operations is shown in table 12 in the annex.

For reasons of completeness and with reference
to fig. 5a, it is indicated that, on a negative re-
sponse to test 504, the continuation of the transforma-
tion process is implemented by a context continuation
5   stage, continue 503, that the negative response to
tests 505, 508, 511 and 513 is itself followed by a
continuation of the transformation process via a con-
text continuation stage, continue 503, and that the
same applies to the continuation of operations after
10 the abovementioned redirection stages 507 and 510 and
insertion stages 512 and 516.

A more detailed description of the method of
standardizing and transforming an object code into a
standardized object code using typed registers as de-
15 scribed in fig. 4b will now be given with reference to
fig. 5b. This implementation mode concerns, more par-
ticularly, a nonlimiting, preferred implementation mode
of stage 601 to reallocate registers by detecting the
original registers used with different types.

20 With reference to the abovementioned fig. 5b, it
is indicated that the abovementioned stage 601 consists
in determining, in a stage 603, the lifetime intervals
marked ID_j of each register r_j. These lifetime inter-
vals, designated "*live range*" or "*webs*", are defined
25 for a register r as a maximum set of partial traces
such that register r is live at all points of these
traces. For a more detailed definition of these con-
cepts, it is useful to refer to the work edited by Ste-
ven S. MUCHNICK entitled "Advanced Compiler Design and
30 Implementation", Section 16.3, Morgan KAUFMANN, 1997.
Stage 603 is designated by the relation:

$$ID_j \longleftrightarrow r_j$$

as claimed in which a corresponding lifetime interval
ID_j is associated with each register r_j.

35 The abovementioned stage 603 is followed by a
stage 604 consisting in determining, at stage 604, the
main data type, marked tp_j, of each lifetime interval

ID_j. The main type of a lifetime interval ID_j, for a register r_j, is defined by the upper bound of the data types stored in this register r_j by the backup instruc-tions <u>store</u> belonging to the abovementioned lifetime

5   interval.

Stage 604 is itself followed by a stage 605 con-sisting in establishing an interference graph between the lifetime intervals as defined above at stages 603 and 604, this interference graph consisting of a non-

10   oriented graph of which each peak consists of a life-time interval, and of which the arcs, marked $a_{j1,j2}$ on fig. 5b, between two peaks ID_{j1} and ID_{j2}, exist if a peak contains a backup instruction addressed to the register of the other peak or vice versa. In fig. 5b, the con-

15   struction of the interference graph is shown symboli-cally, it being possible to implement this construction on the basis of calculation techniques which are known to those skilled in the art. For a more detailed de-scription of the construction of this type of graph, it

20   is useful to refer to the work published by Alfred V. AHO,   Ravi   SETHI   and   Jeffrey   D.   ULLMAN   entitled "Compilers: principles, techniques, and tools", Addi-son-Wesley 1986, Section 9.7.

Following stage 605, the standardization method

25   as shown in fig. 5b consists in translating, in a stage 606, the uniqueness of a data type which is allocated to each register r_j in the interference graph, by adding arcs between all pairs of peaks of the interference graph while two peaks of a pair of peaks do not have

30   the same associated main data type. It is understood that the translation of the character of uniqueness of a data type which is allocated to each register obvi-ously corresponds to translating and taking into ac-count criterion C4 in the interference graph, this cri-

35   terion being mentioned previously in the description. The abovementioned stage 606 is then followed by a stage 607 in which an instantiation of the interference graph is carried out, this instantiation being more

commonly designated as the painting stage of the inter-
ference graph as claimed in the usual techniques. Dur-
ing stage 607, the transformation process assigns to
each lifetime interval ID_{jk} a register number rk, in
5   such a way that two adjacent intervals in the interfer-
ence graph receive different register numbers.

This operation can be implemented on the basis
of any suitable process. As a nonlimiting example, it
is indicated that a preferred process can consist:

10   a)   in choosing a peak of minimum degree in the in-
           terference graph, minimum degree being defined
           as a minimum number of adjacent peaks, and with-
           drawing it from the graph. This stage can be re-
           peated until the graph is empty.

15   b)   Each previously withdrawn peak is reintroduced
           into the interference graph in the inverse order
           of their withdrawal, the last to be removed be-
           ing the first to be reintroduced, and succes-
           sively in the inverse order of the order of
20   withdrawal. Thus the smallest register number
           which is different from the numbers assigned to
           all the adjacent peaks can be assigned to each
           reintroduced peak.

Finally, by stage 602, shown in fig. 4b, the transfor-
25   mation and reallocation process rewrites the access in-
structions to the registers in the code of the subpro-
gram of the relevant applet. Access to a given register
in the corresponding lifetime interval is replaced by
access to a different register, the number of which has
30   been assigned during the instantiation phase, also des-
ignated the painting phase.

A more detailed description of an on-board data-
processing system, making it possible to implement the
management protocol and verification process of a pro-
35   gram fragment or applet as claimed in the subject of
the present invention, and of a development system of
an applet, will now be given with reference to fig. 6.

Regarding the corresponding on-board system with reference 10, it is recalled that this on-board system is a reprogrammable-type system, including the essen-tial components as shown in fig. 1b. The abovementioned on-board system is considered to be interconnected to a terminal by a serial link, the terminal itself being linked, for instance via a local network, if appropri-ate a remote network, to an applet development computer with reference 20. On the on-board system 10 runs a main program which reads and executes the commands which the terminal sends on the serial link. Addition-ally, the standard commands for a microprocessor card, such as for instance the standard commands of the ISO 7816 protocol, can be implemented, and the main program recognizes two additional commands, one for remote loading of an applet, and the other for selecting an applet which has previously been loaded onto the micro-processor card.

In conformity with the subject of the present invention, the structure of the main program is imple-mented in such a way as to include at least one program module for management and verification of a downloaded program fragment, following the protocol for managing a downloaded program fragment as described above in the description with reference to fig. 2.

Additionally, the program module also includes a subprogram module to verify a downloaded program frag-ment, following the verification method as described above in the description with reference to figs. 3a to 3j.

For this purpose, the structure of the memories, in particular the non-writable permanent memory ROM, is modified in such a way as to include in particular, apart from the main program, a protocol management and verification module 17, as mentioned above. Finally, regarding the nonvolatile rewritable memory of EEPROM type, this advantageously includes a directory of ap-plets, marked 18, making it possible to implement the

management protocol and verification process which are
the subjects of the present invention.

With reference to the same fig. 6, it is indi-
cated that the applet development system conforming to
5   the subject of the present invention, in fact making it
possible to transform a traditional object code as men-
tioned above in the description, and satisfying crite-
ria C1+C2+C'3+C'4 in the framework of the Java environ-
ment, into a standardized object code for the same pro-
10   gram fragment, includes, associated with a traditional
Java compiler 21, a code transformation module, marked
22, which proceeds to transform code into standardized
code as claimed in the first and second implementation
modes described above in the description with reference
15   to figs. 4a, 4b and 5a, 5b. It is in fact understood
that, on the one hand, standardization of the original
object code into a standardized object code with
branching instruction with empty stack and into a stan-
dardized code using typed registers, on the other hand,
20   as mentioned previously in the description, makes it
possible to satisfy verification criteria C3 and C4,
which are imposed by the verification method which is
the subject of the present invention.

The code transformation module 22 is followed by
25   a JavaCard transformer 23, which makes it possible to
ensure transmission by a remote or local network to the
terminal and, via the serial link, to the microproces-
sor card 10. Thus the applet development system 20
shown in fig. 6 makes it possible to transform the com-
30   piled class files produced by the Java compiler 21 from
the Java source codes of the applet into class files
which are equivalent, but which observe the additional
constraints C3, C4 which are imposed by the management
protocol and the verification module 17 on board the
35   microprocessor card 10. These transformed class files
are transformed into a downloadable applet on the card
by the standard JavaCard transformer 23.

Various particularly noteworthy components of the set of protocol components, methods and systems which are the subjects of the present invention will now be given for information only.

5   Compared to the verification processes of the prior art as mentioned in the introduction to the description, the verification method which is the subject of the present invention appears noteworthy in that it concentrates the verification effort on the typing

10   properties of the operands which are essential to the security of execution of each applet, i.e. observing the type constraints associated with each instruction and absence of stack overflow. Other verifications do not appear to be essential in terms of security, in

15   particular verification that the code correctly initializes every register before reading it for the first time. On the contrary, the verification method which is the subject of the present invention operates by initializing to zero all the registers from the virtual

20   machine when the method is initialized, to guarantee that reading a non-initialized register cannot compromise the security of the card.

Additionally, the demand imposed by the verification method which is the subject of the present in-

25   vention, as claimed in which the stack must be empty at each branching or branching target instruction, guarantees that the stack is in the same state, empty, after execution of the branching and before execution of the instruction to which the program has branched. This

30   mode of operation guarantees that the stack is in a consistent state, whatever the execution route which is followed through the code of the relevant subprogram or applet. The consistency of the stack is thus guaranteed even in the presence of a branching or branching tar-

35   get. Contrary to the methods and systems of the prior art, in which it is necessary to conserve in random-access memory the type of the stack at each branching target, which necessitates a quantity of random-access

memory proportional to TpxNb, the product of the maxi-
mum size of execution stack which is used and the num-
ber of branching targets in the code, the verification
method which is the subject of the present invention
5   only needs the type of the execution stack at the time
of the instruction during verification, and it does not
keep in memory the type of this stack at other points
of the code. Consequently, the method which is the sub-
ject of the invention is satisfied with a quantity of
10 random-access memory proportional to Tp but independent
of Nb, and consequently of the length of the code of
the subprogram or applet.

The requirement as claimed in criterion C4, as
claimed in which a given register must be used with one
15 and the same type throughout the code of a subprogram,
guarantees that the abovementioned code does not use a
register in an inconsistent way, e.g. by writing a
short integer to it at one point of the program and re-
reading it as an object reference at another point of
20 the program.

In the verification processes which are de-
scribed in the prior art, in particular in the previ-
ously mentioned Java specification entitled "The Java
Virtual Machine Specification", edited by Tim LINDHOLM
25 and Frank YELLIN, to guarantee the consistency of the
abovementioned uses through the branching instructions,
it is necessary to keep in random-access memory a copy
of the table of register types at each branching tar-
get. This operation necessitates a quantity of random-
30 access memory proportional to $T_r x N_b$, where T_r designates
the number of registers used by the subprogram and N_b
the number of branching targets in the code of this
subprogram.

On the contrary, the verification process which
35 is the subject of the present invention operates on a
global table of register types without keeping a copy
at different points of the code in random-access mem-

ory. Consequently, the random-access memory which is
required to implement the verification process is pro-
portional to T_r but independent of N_b, and consequently
of the length of the code of the relevant subprogram.

5   The constraint as claimed in which a given reg-
ister is used with the same type at all points, i.e. at
every instruction of the relevant code, simplifies ap-
preciably and significantly the verification of subpro-
grams. On the contrary, in the verification processes
10   of the prior art, in the absence of such a constraint,
the verification process must establish that the sub-
programs observe a strict stack discipline, and must
verify the body of the subprograms polymorphously re-
garding the type of certain registers.

15   In conclusion, the verification process which is
the subject of the present invention, compared to the
techniques of the prior art, makes it possible, on the
one hand, to reduce the size of the program code which
makes it possible to carry out the verification method,
20   and on the other hand, to reduce the consumption of
random-access memory during the verification opera-
tions, the degree of complexity being of the form
$O(T_p+P_r)$ in the case of the verification process which
is the subject of the present invention, instead of
25    $(O(T_p+T_r) \times N_b)$ for the verification process of the prior
art, while however offering the same guarantees about
the security of execution of the verified code.

      Finally, the process of transforming original
traditional code into standardized code is implemented
30   by localized transformation of the code without trans-
mitting additional information to the verifier compo-
nent, i.e. the microprocessor card or on-board data-
processing system.

      Regarding the method of reallocating registers
35   as described in figs. 4b and 5b, this method differs
from the known methods of the prior art, as described
in particular in US Patents 4,571,678 and 5,249,295, by
the fact that:

−   the register reallocation ensures that the same
         register cannot be assigned to two intervals
         with different main types, which thus guarantees
         that a given register is used with the same type
5   throughout the code; and

−   the existing register allocation algorithms,
         which are described in the abovementioned docu-
         ments, assume a fixed number of registers, and
         attempt to minimize the transfers, called
10   "*spills*", between registers and stack, whereas
         reallocation of registers as claimed in the sub-
         ject of the present invention operates in a
         framework where the total number of registers is
         variable, as a consequence of which there is no
15   purpose in carrying out transfers between regis-
         ters and stacks when a process of minimizing the
         total number of registers is carried out.

         The protocol for managing a program fragment
downloaded onto an on-board system, and the methods of
20   verifying this downloaded program fragment and of
transforming this object code of a downloaded program
fragment respectively, which are the subjects of the
present invention, can of course be implemented in
software.

25   Therefore, the present invention also concerns a
computer program product which can be loaded directly
into the internal memory of a reprogrammable on-board
system, this on-board system making it possible to
download a program fragment consisting of an object
30   code, a series of instructions, executable by the mi-
croprocessor of the on-board system by way of a virtual
machine equipped with an execution stack and with local
registers or variables manipulated via these instruc-
tions so that this object code can be interpreted. The
35   corresponding computer program product includes por-
tions of object code to execute the protocol for manag-
ing a program fragment downloaded onto this on-board
system, as shown in figs. 2 and 6 described above in

the description, when this on-board system is intercon-
nected to a terminal and this program is executed by
the microprocessor of this on-board system by way of
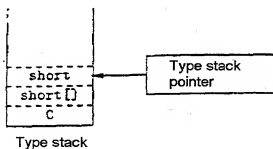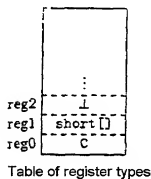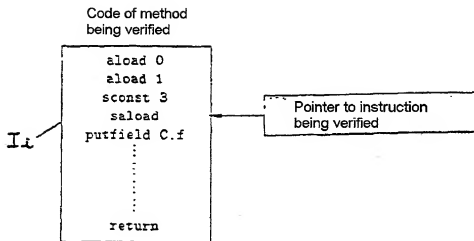the virtual machine.

5   The invention also concerns a computer program
product which can be loaded directly into the internal
memory of a reprogrammable on-board system, such as a
microprocessor card with a rewritable memory, as shown
with reference to fig. 6. This computer program product
10   includes portions of object code to execute the stages
of verifying a program fragment downloaded onto this
on-board system, as shown and described above in the
description, with reference to figs. 3a to 3j. This
verification is executed when this on-board system is
15   interconnected to a terminal and this program is exe-
cuted by the microprocessor of this on-board system via
the virtual machine.

          The invention also concerns a computer program
product; this computer program product includes por-
20   tions of object code to execute the stages of the
method of transforming the object code of a program
fragment into standardized object code for this same
program fragment, as shown in figs. 4a, 4b, 5a, 5b and
6, and described above in the description.

25   The present invention also concerns a computer
program product which is recorded on a medium which can
be used in a reprogrammable on-board system, e.g. a mi-
croprocessor equipped with a rewritable memory, this
on-board system making it possible to download a pro-
30   gram fragment consisting of an object code executable
by this microprocessor, by way of a virtual machine
equipped with an execution stack and local variables or
registers manipulated via these instructions, to allow
interpretation of this object code. The abovementioned
35   computer program product includes, at least, a module
of programs which can be read by the microprocessor of
the on-board system via the virtual machine, to command
execution of a procedure for managing the downloading

of a downloaded program fragment, as shown in fig. 2
and described above in the description, a module of
programs which can be read by the microprocessor via
the virtual machine, to command execution of a proce-
5   dure for verifying, instruction by instruction, the ob-
ject code which makes up this program fragment, as
shown and described in relation to figs. 3a to 3j in
the description above, and a module of programs which
can be read by the microprocessor of this on-board sys-
10   tem via the virtual machine, to command execution of a
downloaded program fragment following or in the absence
of a transformation of the object code of this program
fragment into a standardized object code for this same
program fragment, as shown in fig. 2.

15   The abovementioned computer program product also
includes a module of programs which can be read by the
microprocessor via the virtual machine, to command in-
hibition of execution, on the on-board system, of the
program fragment in the case of an unsuccessful verifi-
20   cation procedure of the abovementioned program frag-
ment, as shown and described above in the description
with reference to fig. 2.

ANNEXES

Code of method
being verified

```
        aload 0
        aload 1
        sconst 3
        saload
Iᵢ    putfield C.f
          ⋮



        return
```

Pointer to instruction
being verified

reg2	⊥
reg1	short []
reg0	C

Table of register types

	short
	short []
	C

Type stack

Type stack
pointer

TABLE 2

Pseudo-code of verifier module

PSEUDO-CODE OF VERIFIER MODULE

Global variables used:

T_r	number of registers declared by current method
T_p	maximum size of stack declared by current method
$tr[T_r]$	table of register types (402 in fig. 4)
$tp[T_p]$	stack type (403 in fig. 4)
pp	stack pointer (404 in fig. 4)
chg	flag indicating whether tr has changed

Initialize $pp \to 0$

Initialize $tp[0] \dots tp[n-1]$ from types of n arguments of method

Initialize $tp[n] \dots tp[T_r-1]$ to $\perp$

Initialize chg to true

While chg is true:

    Reset chg to false

    Position on first instruction of method

    While end of method is not reached:

        If current instruction is target of a branching instruction:

            If $pp \neq 0$, verification fails

        If current instruction is target of a subroutine call:

            If previous instruction continues in sequence, failure

            Take $tp[0] \leftarrow$ ·etaddr and $pp \leftarrow 1$

        If current instruction is an exception handler of class C:

            If previous instruction continues in sequence, failure

            Do $tp[0] \leftarrow C$ and $pp \leftarrow 1$

        If current instruction is a target of different kinds:

            Verification fails

        Determine types $a_1 \dots a_n$ of arguments of instruction

        If $pp < n$, failure (stack overflow)

        For $i = 1, \dots, n$:

            If $tp[pp-n-i-1]$ is not subtype of a_i, failure

        Do $pp \leftarrow pp-n$

        Determine types $r_1, \dots, r_m$ of results of instruction

        If $pp+m > T_p$, failure (stack overflow)

        For $i = 1, \dots, m$, do $tp[pp+i-1] ? r_i$

        Do $pp \leftarrow pp+m$

        If current instruction is a write to a register r:

Determine type *t* of value written to record

Do *tr*[*r*] ←*lower bound* (*t*, *tr*[*r*])

If *tr*[*r*] has changed, do *chg*← true

If current instruction is a branching:

5   If *pp* ≠ 0, verification failure

Advance to next instruction

Return verification success code

10

TABLE T3

```
static short[] meth(short [] table    )
{
    short[] result    = null;
    if  (  table length    >= 2) result = table;
      return table
}
```

TABLE T4

First iteration on method code:

Method code	Table of register types	Stack type
0 : aconst_null	r0: short[] r1: ⊥	
1 : astore 1	r0: short[] r1: ⊥	null
2 : aload 0	r0: short[] r1: null	
3 : arraylength	r0: short[] r1: null	short[]
4 : sconst 2	r0: short[] r1: null	short
5 : if_scmplt 9	r0: short[] r1: null	short | short
7 : aload 0	r0: short[] r1: null	
8 : astore 1	r0: short[] r1: null	short[]
9 : aload 1	r0: short[] | r1: short[]	
10 : areturn	r0: short[] | r1: short[]	short[]

Second iteration on method code:

Method code		Stack type
	r0: short[] | r1: short[]	
0 : aconst_null	r0: short[] | r1: short[]	null
1 : astore 1	r0: short[] | r1: short[]	
2 : aload 0	r0: short[] | r1: short[]	short[]
3 : arraylength	r0: short[] | r1: short[]	short
4 : sconst 2	r0: short[] | r1: short[]	short | short
5 : if_scmplt 9	r0: short[] | r1: short[]	
7 : aload 0	r0: short[] | r1: short[]	short[]
8 : astore 1	r0: short[] | r1: short[]	
9 : aload 1	r0: short[] | r1: short[]	short[]
10 : areturn		

TABLE T5

(a) *Violation of type constraints on arguments of an instruction:*

0 :	aload 0	r0: Object	
1 :	sconst 1	r0: Object	Object
2 :	sadd	r0: Object	Object | short
3 :	areturn		

Error: stack type does not correspond to what instruction expects (short, short)
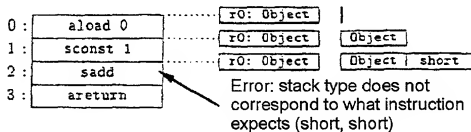
(b) *Inconsistent use of a register:*

0 :	sload 0	r0: short	
1 :	ifeq 6	r0: short	short
4 :	aconst_null	r0: short	
5 :	astore 0	r0: short	null
6 :	aload 0	r0: T	
7 :	areturn	r0: T	T

Error: stack type does not correspond to what instruction expects (Object)
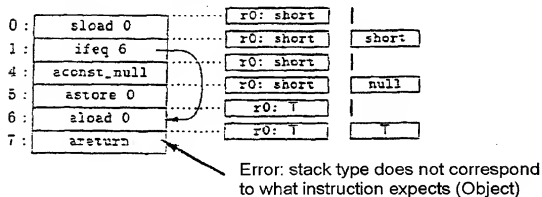
(c) *Branchings introducing inconsistencies at stack level:*

0 :	sload 0	r0: short	
1 :	ifeq 8	r0: short	short
4 :	sconst 1	r0: short	
5 :	goto 9	r0: short	short
8 :	aconst_null		
9 :	areturn		

Error: stack not empty at branching point
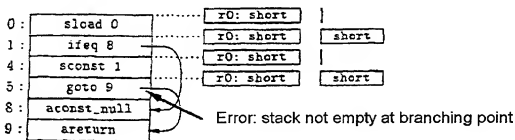
(d) *Stack overflow within a loop:*

0 :	sload 0	r0: short	
1 :	ifeq 11	r0: short	short
4 :	sconst 1	r0: short	
5 :	sinc 0 -1	r0: short	short
8 :	goto 0	r0: short	short
11 :	return		

Error: stack not empty at branching point
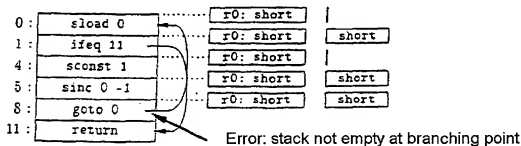
TABLE T6

(a) *Initial code of method, annotated by types of registers and of stack:*

0 :	aload 0	r0: Object	
1 :	ifnull 8	r0: Object	Object
4 :	iconst 1	r0: Object	
5 :	goto 9	r0: Object	int
8 :	iconst 0	r0: Object	
9 :	ineg	r0: Object	int
10 :	istore 0	r0: Object	int
11 :	iload 0	r0: int	
12 :	ireturn	r0: int	int

TABLE T7

(b) *Method code after standardization of stack at branching 5 → 9:*

0 :	aload 0	r0: Object	r1: ⊥	
1 :	ifnull 8	r0: Object	r1: ⊥	Object
4 :	iconst 1	r0: Object	r1: ⊥	
4' :	istore 1	r0: Object	r1: ⊥	int
5 :	goto 8''	r0: Object	r1: int	
8 :	iconst 0	r0: Object	r1: ⊥	int
8' :	istore 1	r0: Object	r1: int	
8" :	iload 1	r0: Object	r1: int	int
9 :	ineg	r0: Object	r1: int	int
10 :	istore 0	r0: int	r1: int	
11 :	iload 0	r0: int	r1: int	int
12 :	ireturn			

TABLE T8

(c) *Method code after reallocation of registers:*

0 :	aload 0		r0: Object	r1: ⊥	
1 :	ifnull 8		r0: Object	r1: ⊥	Object
4 :	iconst 1		r0: Object	r1: ⊥	
4' :	istore 1		r0: Object	r1: ⊥	int
5 :	goto 8''		r0: Object	r1: int	
8 :	iconst 0		r0: Object	r1: int	
8' :	istore 1		r0: Object	r1: ⊥	int
8'' :	iload 1		r0: Object	r1: int	
9 :	ineg		r0: Object	r1: int	int
10 :	istore 1		r0: Object	r1: int	int
11 :	iload 1		r0: Object	r1: int	
12 :	ireturn		r0: Object	r1: int	int

TABLE T9

(a) *Branching target, previous instruction not continuing in sequence:*
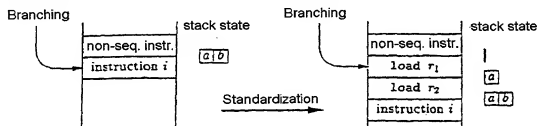
<u>TABLE T10</u>

(b) *Branching target, previous instruction continuing in sequence:*
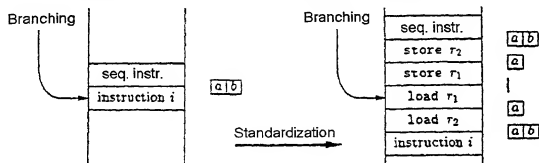
<u>TABLE T11</u>

(c) *Unconditional branching without arguments:*
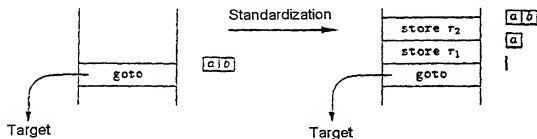
<u>TABLE T12</u>

(c) *Conditional branching with one argument:*
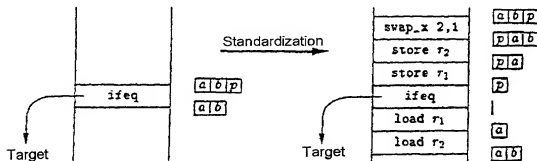
CLAIMS

    1. A protocol for managing a program fragment
downloaded onto a reprogrammable on-board system, such
as a microprocessor card equipped with a rewritable
5   memory, said program fragment consisting of an object
code, a series of instructions, executable by the mi-
croprocessor of the on-board system by way of a virtual
machine equipped with an execution stack and with local
variables or registers manipulated via these instruc-
10   tions and making it possible to interpret this object
code, said on-board system being interconnected to a
terminal, characterized in that this protocol consists
at least, at the level of said on-board system:

    a)   in detecting a command for downloading of this
15   program fragment; and, on a positive response to
            this stage consisting in detecting a downloading
            command,

    b)   in reading the object code constituting this
            program fragment and in temporarily storing this
20   object code;

    c)   in subjecting the whole of the object code
            stored temporarily in memory to a verification
            process, instruction by instruction, this veri-
            fication process consisting at least in a stage
25   of initializing the type stack and the table of
            register types representing the state of said
            virtual machine at the start of the execution of
            the temporarily stored object code and in a suc-
            cession of stages of verification, instruction
30   by instruction, by discerning the existence, for
            each current instruction, of a target, branch-
            ing-instruction target, target of an exception-
            handler call, or target of a subroutine call,
            and in a verification and an updating of the ef-
35   fect of said current instruction on the type
            stack and on the table of register types, and,
            in the event of a successful verification of
            said object code,

d)   in recording the downloaded program fragment in
        a directory of available program fragments, and;
        in the event of an unsuccessful verification of
        said object code,

5   e)   in inhibiting execution, on said on-board sys-
        tem, of said program fragment.

        2. The protocol as claimed in claim 1, charac-
terized in that said stage e) of inhibiting the execu-
tion consists:

10 f)   in deleting the momentarily recorded program
        fragment, when omitting to record the latter in
        said directory of available program fragments,
        and

g)   in sending an error code to said reader.

15   3.The protocol as claimed in claim 1 or 2, char-
acterized in that, on a negative response to said stage
a) consisting in detecting a downloading command, this
consists:

b') ·   in detecting a command to select an available
20   program fragment from a directory of program
        fragments; and, ·on a positive response to this
        stage, consisting in detecting a command to se-
        lect an available program fragment;·

c')   in calling said selected available program frag-
25   ment;

d')   ·in executing said called available program frag-
        ment via the virtual machine, with no dynamic
        verification of variable types, access rights to
        the objects which are manipulated by the called
30   available program fragment, or overflow of the
        execution stack when each instruction is exe-
        cuted, and, on a negative response to this stage
        consisting in detecting a command to select an
        available program fragment,

35 e')   in proceeding to process the standard commands
        of the on-board system.

        4. A method ·of verifying a program fragment
downloaded onto a reprogrammable on-board system, such

as a microprocessor card equipped with a rewritable
memory, said program fragment consisting of an object
code and including at least one subprogram, a series of
instructions, by the microprocessor of the on-board
system by way of a virtual machine equipped with an
execution stack and with operand registers manipulated
by these instructions, and making it possible to inter-
pret this object code, said on-board system being in-
terconnected to a reader, characterized in that said
method, following the detection of a downloading com-
mand and the storage of said object code constituting
this program fragment in said rewritable memory, con-
sists, for each subprogram:

α)   in carrying out a stage of initializing the type
         stack and the table of register types by data
         representing the state of the virtual machine at
         the start of the execution of the temporarily
         stored object code;

β)   in carrying out a verification of said temporar-
         ily stored object code instruction by instruc-
         tion, by discerning the existence, for each cur-
         rent instruction, of a target, a branching-
         instruction target, a target of an exception-
         handler call or a target of a subroutine call;

γ)   in carrying out a verification and an updating
         of the effect of said current instruction on the
         data types of said type stack and of said table
         of register types, on the basis of the existence
         of a branching-instruction target, of a target
         of a subroutine call or of a target of an excep-
         tion-handler call, said verification being suc-
         cessful when the table of register types is not
         modified in the course of a verification of all
         the instructions, and the verification process
         being carried out instruction by instruction un-
         til the table of register types is stable, with

no modification present, the verification process being interrupted otherwise.

5. The verification method as claimed in claim 4, characterized in that the variable types which are

5   manipulated during the verification process include at least:

–   class identifiers corresponding to object classes which are defined in the program fragment;

10   –   numeric variable types including at least a type short, an integer coded on p bits, and a type retaddr for the return address of a jump instruction JSR;

–   a type null relating to references of null objects;

15

–   a type object relating to objects;

–   a first specific type $\bot$, representing the intersection of all the types and corresponding to the value 0, nil;

20   –   a second specific type $\top$, representing the union of all the types and corresponding to any type of value.

6. Method as claimed in claim 5, characterized in that all said variable types verify a subtyping re-

25   lation:

object ε $\top$;

short, retaddr ε $\top$;

$\bot$ ε null, short, retaddr.

7. The method as claimed in one of claims 4 to

30   6, characterized in that when said current instruction is the target of a branching instruction, said verification method consists in verifying that the type stack is empty, the verification process being continued for the following instruction in the case of a positive

35   verification, and the verification process failing and the program fragment being rejected otherwise.

8. The method as claimed in one of claims 4 to 7, characterized in that when said current instruction is the target of a subroutine call, said verification process verifies that the previous instruction is an unconditional branching, a subroutine return or a raising of an exception, said verification process, in the case of a positive verification, proceeding to reupdate the stack of variable types by an entity of retaddr type, the return address of the subroutine, and the verification process failing and the program fragment being rejected otherwise.

9. The method as claimed in one of claims 4 to 8, characterized in that when the current instruction is the target of an exception handler, said verification process verifies that the previous instruction is an unconditional branching, a subroutine return or a raising of an exception, said verification process, in the case of a positive verification, proceeding to reupdate the type stack by entering the exception type, and the verification process failing and the program fragment being rejected otherwise.

10. The method as claimed in one of claims 4 to 9, characterized in that when the current instruction is the target of multiple incompatible branchings, the verification process fails and the program fragment is rejected.

11. The method as claimed in one of claims 4 to 10, characterized in that when the current instruction is not the target of any branching, the verification process continues by passing to an update of the type stack.

12. The method as claimed in one of claims 4 to 11, characterized in that the stage of verification of the effect of the current instruction on the type stack includes, at least:

–   a stage of verifying that the type execution stack includes at least as many entries as the current instruction includes operands;

- a stage of unstacking and of verifying that the types of the entries at the top of the stack are subtypes of the types of the operands of the op- erands of this instruction;

5 - a stage of verifying the existence of a suffi- cient memory space on the type stack to proceed to stack the results of the current instruction;

- a stage of stacking on the stack data types which are assigned to these results.

10   13. The method as claimed in claim 12, charac- terized in that when the current instruction is an in- struction to read a register of address n, the verifi- cation process consists:

- in verifying the data type of the result of this
15   reading, by reading the entry n in the table of register types;

- in determining the effect of the current in- struction on the type stack by unstacking the entries of the stack corresponding to the oper-
20   ands of this current instruction and by stacking the data type of this result.

        14. The method as claimed in claim 12, charac- terized in that when the current instruction is an in- struction to write to a register of address m, the
25 verification process consists:

- in determining the effect of the current in- struction on the type stack and the type t of the operand which is written in this register of address m;

30 - in replacing the type entry of the table of reg- ister types at address m by the type immediately above the previously stored type and above the type t of the operand which is written in this register of address m.

15. A method of transforming an object code of a
program fragment, in which the operands of each in-
struction belong to the data types manipulated by this
instruction, the execution stack does not exhibit any

5   overflow phenomenon, for each branching instruction,
the type of the stack variables at this branching is
the same as of the targets of this branching, into a
standardized object code for this same program frag-
ment, in which the operands of each instruction belong

10   to the data types manipulated by this instruction, the
execution stack does not exhibit any overflow phenome-
non, the execution stack is empty at each branching in-
struction and at each branching-target instruction,
characterized in that this method consists, for all the

15   instructions of said object code:

     –   in annotating each current instruction with the
             data type of the stack before and after execu-
             tion of this instruction, the annotation data
             being calculated by means of an analysis of the

20   data stream relating to this instruction;

     –   in detecting, within said instructions and
             within each current instruction, the existence
             of branchings, or respectively of branching-
             targets, for which said execution stack is not

25   empty, the detection operation being carried out
             on the basis of the annotation data of the type
             of stack variables allocated to each current in-
             struction, and in the presence of detection of a
             non-empty execution stack,

30   –   in inserting instructions to transfer stack
             variables on either side of these branchings or
             of these branching targets, respectively in or-
             der to empty the contents of the execution stack
             into temporary registers before this branching

35   and to reestablish the execution stack from said
             temporary registers after this branching, and in
             not inserting any transfer instruction other-
             wise, making it possible to obtain a standard-

ized object code for this same program fragment,
in which the execution stack is empty at each
branching instruction and at each branching-
target instruction, in the absence of any modi-
5   fication to the execution of said program frag-
ment.

16. A method of transforming an object code of a
program fragment, in which the operands of each in-
struction belong to the data types manipulated by this
10   instruction, and an operand of given type written into
a register by an instruction of this object code is re-
read from this same register by another instruction of
this object code with the same given data type, into a
standardized object code for this same program frag-
15   ment, in which the operands of each instruction belong
to the data types manipulated by this instruction, the
same data type being allocated to the same register
throughout said standardized object code, characterized
in that this method consists, for all the instructions
20   of said object code:

-       in annotating each current instruction with the
        data type of the registers before and after exe-
        cution of this instruction, the annotation data
        being calculated by means of an analysis of the
25   data stream relating to this instruction;
-       in carrying out a reallocation of the registers,
        by detecting the original registers employed
        with different types, by dividing these original
        registers into separate standardized registers,
30   one standardized register for each data type
        used, and reupdating the instructions which ma-
        nipulate the operands which use said standard-
        ized registers.

17. The method as claimed in claim 15, charac-
35   terized in that the stage consisting in detecting,
within said instructions and within each current in-
struction, the existence of branchings, or respectively
of branching targets, for which the execution stack is

not empty, consists, following detection of each corre-
sponding instruction of rank i:

–   in associating with each instruction of rank i a
        set of new registers, one new register being as-
        sociated with each stack variable which is ac-
        tive at this instruction;

–   in examining each detected instruction of rank i
        and in discerning the existence of a branching
        target or branching, respectively, and, in the
        case where the instruction of rank i is a
        branching target and that the execution stack at
        this instruction is not empty,

•   for every preceding instruction, of rank
        i-1, consisting of a branching, a rais-
        ing of an exception or a program return,
        the detected instruction of rank i being
        accessible only by a branching,

••   in inserting a set of load instructions
        load to load from the set of new regis-
        ters before said detected instruction of
        rank i, with redirection of all branch-
        ings to the detected instruction of rank
        i to the first inserted load instruction
        load; and

•   for every preceding instruction, of rank
        i-1, continuing in sequence, the de-
        tected instruction of rank i being ac-
        cessible simultaneously by a branching
        and from the preceding instruction of
        rank i-1,

••   in inserting a set of backup instruc-
        tions store to back up to the set of new
        registers before the detected instruc-
        tion of rank i, and a set of load in-
        structions load to load from this set of
        new registers, with redirection of all
        the branchings to the detected instruc-

tion of rank i to the first inserted
load instruction load, and, in the case
where said detected instruction of rank
i is a branching to a given instruction,

• for every detected instruction of rank i
consisting of an unconditional branch-
ing,

•• in inserting, before the detected in-
struction of rank i, multiple backup in-
structions store, a backup instruction
being associated with each new register;
and

• for every detected instruction of rank i
consisting of a conditional branching,
and for a number m > 0 of operands ma-
nipulated by this conditional branching
instruction,

•• in inserting, before this detected in-
struction of rank i, a permutation in-
struction, swap-x, at the top of the
execution stack of the m operands of the
detected instruction of rank i and the n
following values, this permutation op-
eration making it possible to collect at
the top of the execution stack the n
values to be backed up in the set of new
registers, and

•• in inserting, before the instruction of
rank i, a set of backup instructions
store to back up to the set of new reg-
isters, and

•• in inserting, after the detected in-
struction of rank i, a set of load in-
structions load to load from the set of
new registers.

18. The method as claimed in claim 16, charac-
terized in that the stage consisting in reallocating

registers by detecting the original registers employed
with different types consists:

- in determining the lifetime intervals of each
  register;

5 - in determining the main data type of each life-
  time interval, the main data type of a lifetime
  interval j for a register r being defined by the
  upper bound of the data types stored in this
  register r by the backup instructions store be-

10 longing to the lifetime interval j;

- in establishing an interference graph between
  the lifetime intervals, this interference graph
  consisting of a non-oriented graph of which each
  peak consists of a lifetime interval, and of

15 which the arcs between two peaks j_1 and j_2 exist
  if a peak contains a backup instruction ad-
  dressed to the register of the other peak or
  vice versa;

- in translating the uniqueness of a data type

20 which is allocated to each register in the in-
  terference graph, by adding arcs between all
  pairs of peaks of the interference graph while
  two peaks of a pair of peaks do not have the
  same associated main data type;

25 - in carrying out an instantiation of the inter-
  ference graph, by assigning to each lifetime in-
  terval a register number, in such a way that
  different register numbers are assigned to two
  adjacent life intervals in the interference

30 graph.

19. An on-board system which can be reprogrammed
by downloading program fragments, including at least
one microprocessor, one random-access memory, one in-
put/output module, one electrically reprogrammable non-

35 volatile memory and one permanent memory, in which are
installed a main program and a virtual machine which
makes it possible to execute the main program and at
least one program fragment using said microprocessor,

characterized in that said on-board system includes at
least one program module to manage and verify a down-
loaded program fragment, said management and verifica-
tion program module being installed in the permanent
5   memory.

20. An on-board system which can be reprogrammed
by downloading program fragments, including at least
one microprocessor, one random-access memory, one in-
put/output module, one electrically reprogrammable non-
10   volatile memory and one permanent memory, in which are
installed a main program and a virtual machine which
makes it possible to execute the main program and at
least one program fragment using said microprocessor,
characterized in that said on-board system includes at
15   least one program module to manage and verify a down-
loaded program fragment in accordance with the protocol
for managing a downloaded program fragment as claimed
in one of claims 1 to 3, said management and verifica-
tion program module being installed in the permanent
20   memory.

21. The on-board system as claimed in claim 20,
characterized in that it includes at least one subpro-
gram module to verify a downloaded program fragment, in
accordance with the verification process as claimed in
25   one of claims 4 to 14.

22. A method of transforming an object code of a
program fragment, in which the operands of each in-
struction belong to the data types manipulated by this
instruction, the execution stack does not exhibit any
30   overflow phenomenon, for each branching instruction,
the type of stack variables at this branching is the
same as at the targets of this branching, and an oper-
and of given type written to a register by an instruc-
tion of this object code is reread from this same reg-
35   ister by another instruction of this object code with
the same given data type, into a standardized object
code for this same program fragment, in which the oper-
ands of each instruction belong to the data types ma-

nipulated by this instruction, the execution stack does
not exhibit overflow phenomenon, the execution stack is
empty at each branching instruction and at each branch-
ing-target instruction, the same data type being as-
5   signed to the same register throughout said standard-
ized object code, characterized in that said conversion
system includes, at least, installed in the working
memory of a development computer or workstation, a pro-
gram module to transform this object code into a stan-
10   dardized object code in accordance with the method as
claimed in one of claims 15 to 18, making it possible
to generate a standardized object code for said program
fragment, satisfying the criteria for verifying this
downloaded program fragment.

15   23. A computer program product which can be
loaded directly into the internal memory of a repro-
grammable on-board system, such as a microprocessor
card equipped with a rewritable memory, this on-board
system making it possible to download a program frag-
20   ment consisting of an object code, a series of instruc-
tions, executable by the microprocessor of the on-board
system by way of a virtual machine equipped with an
execution stack and with local variables or registers
manipulated via these instructions and making it possi-
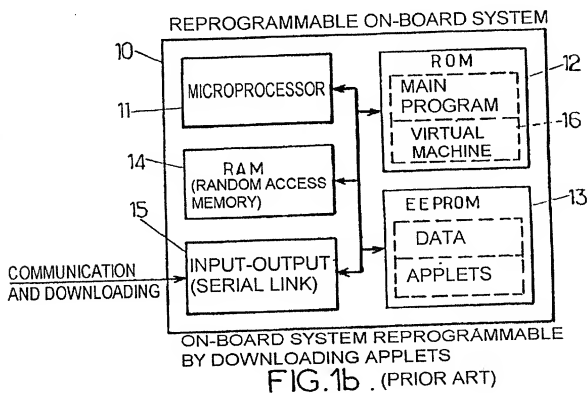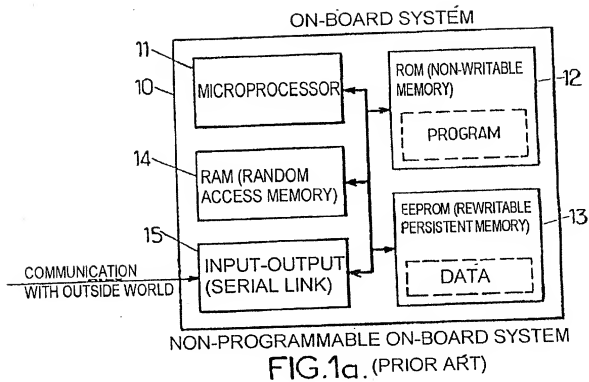25   ble to interpret this object code, this computer pro-
gram product including portions of object code to exe-
cute the protocol for managing a downloaded program
fragment on this on-board system as claimed in one of
claims 1 to 3, when this on-board system is intercon-
30   nected to a terminal and this program is executed by
the microprocessor of this on-board system by way of
said virtual machine.

            24. A computer program product which can be
loaded directly into the internal memory of a repro-
35   grammable on-board system, such as a microprocessor
card equipped with a rewritable memory, this on-board
system making it possible to download a program frag-
ment consisting of an object code, a series of instruc-

tions, executable by the microprocessor of the on-board
system by way of a virtual machine equipped with an
execution stack and with operand registers manipulated
via these instructions and making it possible to inter-

5   pret this object code, this computer program product
including portions of object code to execute the stages
of verifying a program fragment downloaded onto this
on-board system as claimed in one of claims 4 to 14,
when this on-board system is interconnected to a termi-

10 nal and this program is executed by the microprocessor
of this on-board system by way of said virtual machine.

        25. A computer program product including por-
tions of object code to execute stages of the method of
transforming an object code of a downloaded program

15 fragment into a standardized object code for this same
program fragment as claimed in one of claims 15 to 18.

        26. A computer program product which is recorded
on a medium which can be used in a reprogrammable on-
board system, such as a microprocessor card equipped

20 with a rewritable memory, this on-board system making
it possible to download a program fragment consisting
of an object code, a series of instructions, executable
by the microprocessor of the on-board system by way of
a virtual machine equipped with an execution stack and

25 with local variables or registers manipulated via these
instructions and making it possible to interpret this
object code, this computer program product including,
at least:

-       program resources which can be read by the mi-
30   croprocessor of this on-board system via said
        virtual machine, to command execution of a pro-
        cedure for managing the downloading of a down-
        loaded program fragment;

-       program resources which can be read by the mi-
35   croprocessor of this on-board system via said
        virtual machine, to command execution of a pro-
        cedure for verifying, instruction by instruc-

tion, the object code which makes up said pro-
gram fragment;

–   program resources which can be read by the mi-
croprocessor of this on-board system via said
5   virtual machine, to command execution of a down-
loaded program fragment following or in the ab-
sence of a conversion of the object code of this
program fragment into a standardized object code
for this same program fragment.

10   27. The computer program product as claimed in
claim 26, additionally including program resources
which can be read by the microprocessor of this on-
board system via said virtual machine, to command inhi-
bition of execution, on said on-board system, of said
15 program fragment in the case of an unsuccessful verifi-
cation procedure of this program fragment.

ON-BOARD SYSTEM

MICROPROCESSOR

ROM (NON-WRITABLE MEMORY)

PROGRAM

RAM (RANDOM ACCESS MEMORY)

EEPROM (REWRITABLE PERSISTENT MEMORY)

DATA

COMMUNICATION WITH OUTSIDE WORLD

INPUT-OUTPUT (SERIAL LINK)

NON-PROGRAMMABLE ON-BOARD SYSTEM

FIG.1a. (PRIOR ART)

REPROGRAMMABLE ON-BOARD SYSTEM

MICROPROCESSOR

ROM

MAIN PROGRAM

VIRTUAL MACHINE

RAM (RANDOM ACCESS MEMORY)

EEPROM

DATA

APPLETS

COMMUNICATION AND DOWNLOADING

INPUT-OUTPUT (SERIAL LINK)

ON-BOARD SYSTEM REPROGRAMMABLE BY DOWNLOADING APPLETS

FIG.1b . (PRIOR ART)

FIG.2. MANAGEMENT PROTOCOL

READ COMMAND — 100a

COMMAND= DOWNLOAD ? — 100b

YES → LOAD APPLET CODE — 101 → VERIFY LOADED CODE — 102

VERIFICATION SUCCESSFUL? — 103

103a + → ADD APPLET TO DIRECTORY — 104 → ACKNOWLEDGE RECEPTION — 105

103b − → DELETE APPLET — 106 → ERROR CODE — 107

103c

NO

COMMAND= SELECT APPLET — 108

YES → SEARCH FOR APPLET IN DIRECTORY — 109 → EXECUTE APPLET WITHOUT DYNAMIC VERIFICATIONS — 110

NO → PROCESS STANDARD COMMANDS — 111

FIG.3a.
VERIFICATION
METHOD

- 101
- 102
- INITIALIZE STACK AND TABLE OF REGISTER TYPES — 200
- 200a POSITION i ON FIRST INSTRUCTION OF OBJECT CODE => Ii
- 201 — VERIFY INSTRUCTION Ii Ii= CIB ? Ii = CEM ? Ii=CSR ?
- 204 — i= i+1
- VERIFY EFFECT OF Ii ON STACK AND TABLE OF REGISTER TYPES — 202
- 203 Ii= LAST INSTRUCTION OF OBJECT CODE ?
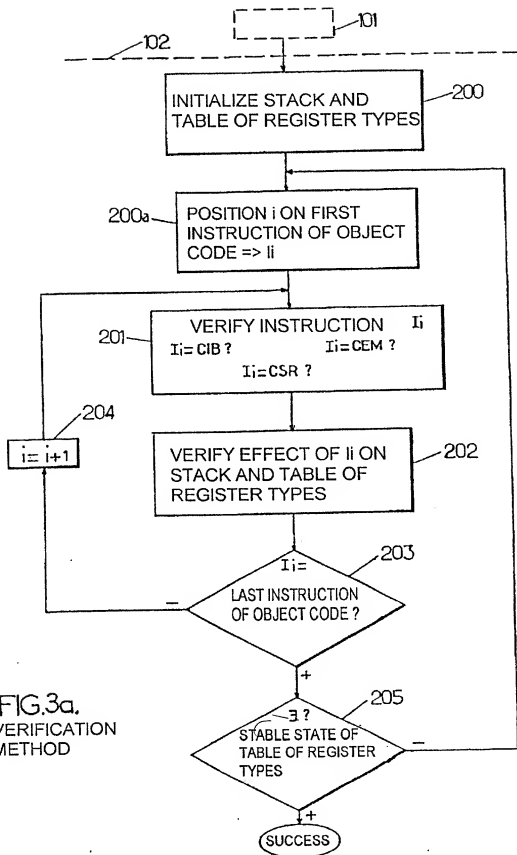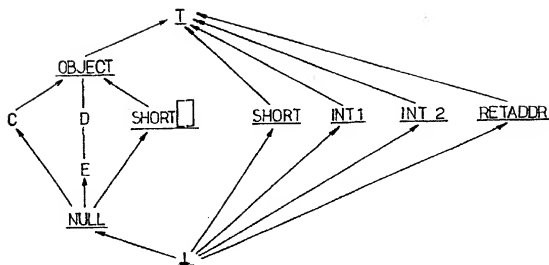- 205 ∃ ? STABLE STATE OF TABLE OF REGISTER TYPES
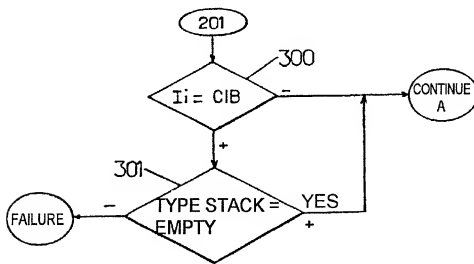- SUCCESS

FIG.3b. DATA TYPES AND SUBTYPING
RELATION

FIG.3c. VERIFICATION: MANAGEMENT OF BRANCHING
INSTRUCTION
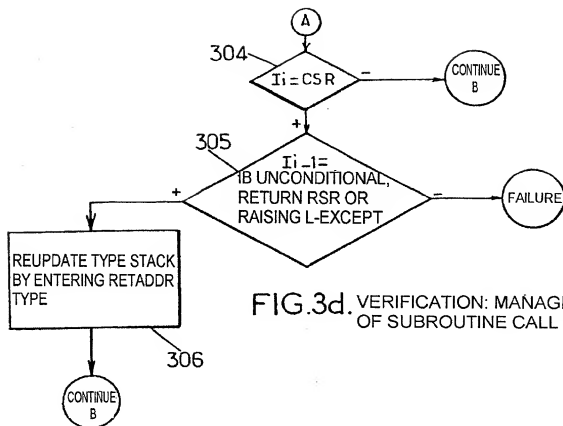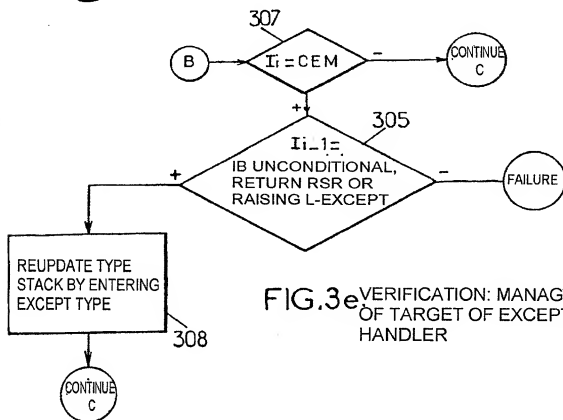
FIG.3d. VERIFICATION: MANAGEMENT OF SUBROUTINE CALL

FIG.3e VERIFICATION: MANAGEMENT OF TARGET OF EXCEPTION HANDLER

FIG.3f. VERIFICATION: MANAGEMENT OF TARGET OF INCOMPATIBLE BRANCHINGS

FIG.3g. VERIFICATION: MANAGEMENT OF ABSENCE OF BRANCHING TARGET

FIG.3h. VERIFICATION: EFFECT OF CURRENT INSTRUCTION ON TYPE STACK

FIG.3i. VERIFICATION: MANAGEMENT OF INSTRUCTION TO READ REGISTER

FIG.3j. VERIFICATION: MANAGEMENT OF INSTRUCTION TO WRITE REGISTER

The flowchart contains the following elements:

- Circle labeled **F**
- Decision diamond 407:
  Ii =
  IWm
  REGISTER OF
  ADDRESS m
- Circle labeled **CONTINUE 204** (via − branch)
- Box 408 (via + branch):
  DETERMINE EFFECT OF li ON TYPE STACK AND TYPE t OF OPERAND Opm
- Box 409:
  REPLACE ENTRY m OF TYPE OF TABLE OF REGISTER TYPES BY TYPE ABOVE PREVIOUSLY STORED TYPE AND OF TYPE t

FIG.4a.

METHOD OF TRANSFORMING CODE TO
STANDARDIZED CODE WITH BRANCHING
INSTRUCTION WITH EMPTY
STACK

FIG.4b. METHOD OF TRANSFORMING CODE TO STANDARDIZED CODE USING TYPE REGISTERS

FIG. 5a.
TRANSFORMATION METHOD
ACCORDING TO FIG. 4a

FIG.5b.

TRANSFORMATION METHOD
ACCORDING TO FIG. 4b

FIG.6.

ON-BOARD SYSTEM

MICROPROCESSOR

ROM (NON-WRITABLE MEMORY)

PROGRAM

RAM (RANDOM ACCESS MEMORY)

EEPROM (REWRITABLE PERSISTENT MEMORY)

DATA

COMMUNICATION WITH OUTSIDE WORLD

INPUT-OUTPUT (SERIAL LINK)

NON-PROGRAMMABLE ON-BOARD SYSTEM

FIG.1a. (PRIOR ART)

REPROGRAMMABLE ON-BOARD SYSTEM

MICROPROCESSOR

ROM

MAIN PROGRAM

VIRTUAL MACHINE

RAM (RANDOM ACCESS MEMORY)

EEPROM

DATA

APPLETS

COMMUNICATION AND DOWNLOADING

INPUT-OUTPUT (SERIAL LINK)

ON-BOARD SYSTEM REPROGRAMMABLE BY DOWNLOADING APPLETS

FIG.1b . (PRIOR ART)

FIG.2. MANAGEMENT PROTOCOL

FIG.3a.
VERIFICATION
METHOD

INITIALIZE STACK AND
TABLE OF REGISTER TYPES — 200

POSITION i ON FIRST
INSTRUCTION OF OBJECT
CODE => Ii
200a

VERIFY INSTRUCTION   Ii
Ii= CIB ?   Ii= CEM ?
Ii=CSR ?
201

i= i+1
204

VERIFY EFFECT OF Ii ON
STACK AND TABLE OF
REGISTER TYPES — 202

Ii= — 203
LAST INSTRUCTION
OF OBJECT CODE ?

∃ ? — 205
STABLE STATE OF
TABLE OF REGISTER
TYPES

SUCCESS

FIG.3b. DATA TYPES AND SUBTYPING RELATION

FIG.3c. VERIFICATION: MANAGEMENT OF BRANCHING INSTRUCTION

FIG.3d. VERIFICATION: MANAGEMENT OF SUBROUTINE CALL

FIG.3e VERIFICATION: MANAGEMENT OF TARGET OF EXCEPTION HANDLER

FIG.3f. VERIFICATION: MANAGEMENT OF TARGET OF INCOMPATIBLE BRANCHINGS

FIG.3g. VERIFICATION: MANAGEMENT OF ABSENCE OF BRANCHING TARGET

FIG.3h. VERIFICATION: EFFECT OF CURRENT INSTRUCTION ON TYPE STACK
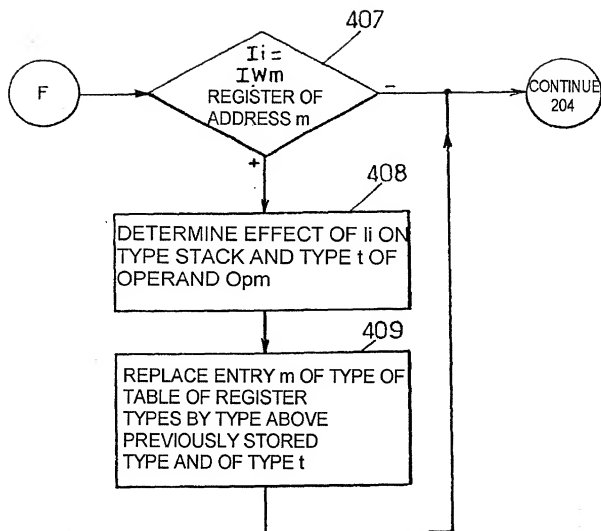
FIG.3i. VERIFICATION: MANAGEMENT OF INSTRUCTION TO READ REGISTER

FIG.3j. VERIFICATION: MANAGEMENT OF INSTRUCTION TO WRITE REGISTER

Flowchart contents:

F →

407: Ii = IWm REGISTER OF ADDRESS m

− → CONTINUE 204

+ →

408: DETERMINE EFFECT OF li ON TYPE STACK AND TYPE t OF OPERAND Opm

409: REPLACE ENTRY m OF TYPE OF TABLE OF REGISTER TYPES BY TYPE ABOVE PREVIOUSLY STORED TYPE AND OF TYPE t

FIG.4a.

METHOD OF TRANSFORMING CODE TO
STANDARDIZED CODE WITH BRANCHING
INSTRUCTION WITH EMPTY
STACK

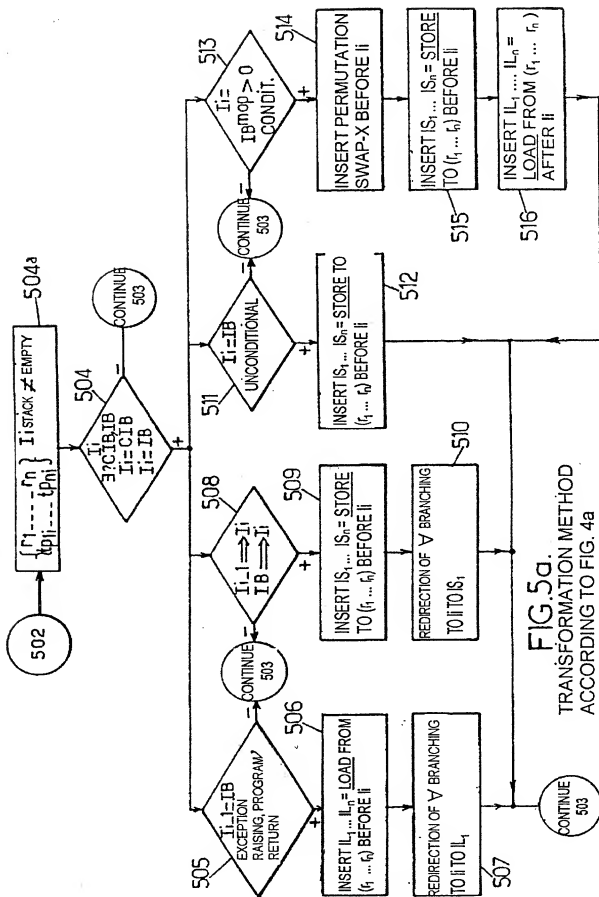FIG.4b. METHOD OF TRANSFORMING CODE TO
STANDARDIZED CODE USING TYPE REGISTERS
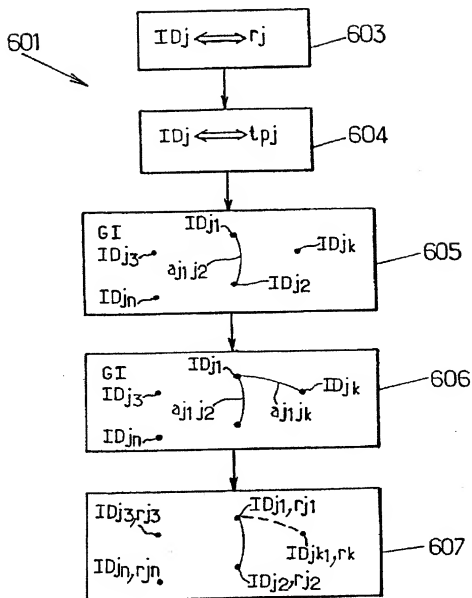
FIG.5a.
TRANSFORMATION METHOD
ACCORDING TO FIG. 4a
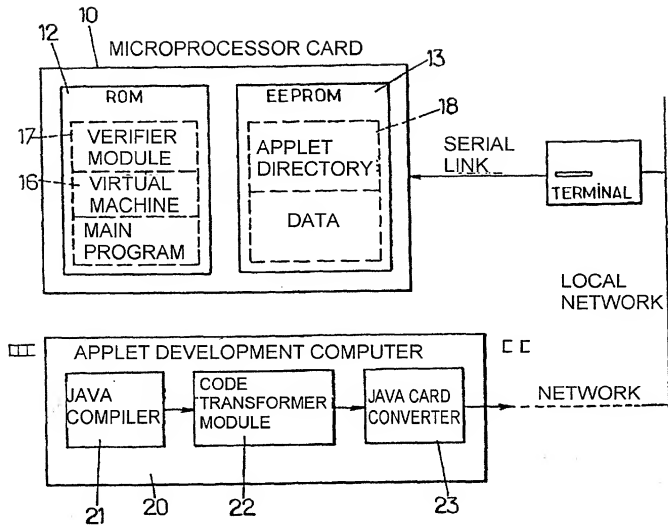
FIG.5b.
TRANSFORMATION METHOD
ACCORDING TO FIG. 4b

FIG.6.